2002

Steiner tree and interconnect optimization in VLSI design.

Jiang. Zhao
University of Windsor

Steiner Tree and Interconnect Optimization in VLSI Design

by

Jiang Zhao

A Thesis
Submitted to the Faculty of Graduate Studies and Research
Through the Department of Electrical Engineering
In Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
At the University of Windsor

Windsor, Ontario, Canada
2002

Abstract

One of the key problems in VLSI interconnect design is the topology construction of signal nets with minimum cost. Steiner tree problem is to find the tree structure that connects all pins of a signal net such that the wire length can be minimized. Since Steiner tree problem is NP-hard, many different heuristics and approximation algorithms have been derived to deal with this problem. However, in VLSI design automation the routing is performed in the presence of obstacles, such as logic cells, where the wires of the net must not intersect. Most of the previous heuristics deal with problems under the assumption that wires do not cross any obstacles.

In this study, a class of probabilistic approaches to Steiner tree problem has been extensively explored, which is able to construct Steiner tree with good performance in term of wire length or speed, and able to solve the problem in presence of obstacles. Probabilistic model and a series of algorithms based on it have been established and implemented. Extensive experiments conducted on both small- and large-size problems have been designed to show the performance comparison with the state-of-the-art algorithm.

How to deal with blockage and congestion in probabilistic algorithms has been discussed. Also, an optimization algorithm has also been presented, which improves wire length from the results of probabilistic algorithms as well as any other algorithms. In addition, the potential advantages with our approaches are also discussed for further applications.

Acknowledgement

I would like to sincerely thank my supervisor Dr. Chunhong Chen for his guidance and encouragement during this project. He cheerfully spent many hours introducing me to probabilistic approaches, and discussing algorithms, their implementation and experiment results.

I would also like to express my gratitude to my examination committee members Dr. W. C. Miller, Dr. A. Jaekel, and the department head Dr. S. Erfani for their invaluable advice.

Table of Contents

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

Chapter 1 Introduction 1

1.1 An Overview 1

1.2 Scope of Work and Objective 2

1.3 Organization of the Thesis 2

Chapter 2 Minimum Spanning Tree and Steiner Tree 4

2.1 Minimum Spanning Tree 4

2.1.1 Definition 4

2.1.2 Algorithms 4

2.2 Steiner Tree 5

2.2.1 Important Results 5

2.2.1.1 Hanan Grid 5

2.2.1.2 Huang's Theorm 6

2.2.2 Heuristics 6

2.2.2.1 MST Embeddings 6

2.2.2.2 1-Steiner Heuristics 7

Chapter 3 Probabilistic Approaches 8

3.1 Probabilistic Model 8

3.1.1 Grid Graph 8

3.1.2 Model 9

3.1.3 Probability Matrix 16

3.2 Probabilistic Algorithms 17

3.2.1 Pure Probabilistic Algorithm 17
3.2.2 MST Combined Algorithm 19
3.2.3 Non-Deterministic Algorithm 19
3.3 Blockages and Congestion 21

Chapter 4 Software Design and Implementation 23
4.1 Union Find Data Structure 23
4.2 Loop Detection 24
4.3 Delete Redundant edges 25

Chapter 5 Optimization Algorithm 26
5.1 Directed Tree 26
5.2 Loop Detection 28
5.3 Edge Replacement 28

Chapter 6 Experiment and Discussion 30
6.1 Small-Size Examples 30
6.2 Random Examples 33
6.3 Discussion 37

Chapter 7 Conclusion and Suggestion for Future Work 38

References 39

Appendix A 42

Appendix B 68

Vita Auctoris 83

List of Tables

Table		Page
Table 3.1	$F(m, n)$ function	14
Table 3.2	$G(m, n)$ function	15
Table 6.1	Comparison of different algorithms on three small-size examples	34

List of Figures

Figure		Page
2.1	Hanan grid	6
2.2	MST embeddings algorithm example	7
2.3	1-Steiner heuristic example	7
3.1	The grid graph for a set of three points	9
3.2	An optimal Steiner tree of Fig 3.1	9
3.3	Probabilistic analysis for the segments through which the shortest paths between points (p_i and p_j) pass	10
3.4	The graphic representations for $F(m, n)$ and log $F(m, n)$	15
4.1	Find and union examples	24
5.1	Directed tree example	26
5.2	Merging two trees	27
5.3	Optimization algorithm example	28
6.1	The result on an example with $N = 5$ using pure probabilistic algorithm	31
6.2	The result on an example with $N = 11$ using pure probabilistic algorithm (Note that replacing R_1 and C_1 with R_2 can lead to a better result)	32
6.3	Comparison of the results	32
6.4	Wire length comparison for different algorithms on small-size problems	35
6.5	Execution time comparison for different algorithms on small-size problems	35
6.6	Wire length comparison for different algorithms on large-size problems	36
6.7	Execution time comparison for different algorithms on large-size problems	36

Chapter 1

Introduction

1.1 An Overview

Steiner tree, named after Jacob Steiner, a great scientist in the 19th century, is a tree in a distance graph that spans a given subset of vertices with the minimal total distance on its edges. The Steiner tree problem has a wide variety of applications in the area of telecommunication network design as well as printed circuit board design layout. Especially for the last four decades due to its application in VLSI design, the Steiner tree problem has attracted considerable research interest and has been widely studied by the researchers both in VLSI design community and in the area of computer science.

In VLSI design automation, a fundamental step is routing a net, which connects a set of terminals with minimum-length metal wires. If wires are restricted to the horizontal and vertical directions, which are a popular case in VLSI design, this problem is called *rectilinear Steiner tree* (RST). In general, RST may contain, in addition to the pins of the net, some other points (i.e., Steiner points). In particular, RST without Steiner points is called the *rectilinear minimum spanning tree* (RMST) that has been well studied [1]. While RST can generally lead to better results than RMST in terms of wire length, it has been shown that the RST problem is NP-complete [2]. Therefore numerous heuristics have been studied toward optimal or near-optimal solutions (e.g., [3-16]). One of the well-known algorithms with good performance is the *Batched Iterated 1-Steiner* (BI1S) heuristic due to *Kahng* and *Robins* [13].

More generally, in VLSI design automation the routing is performed in the presence of obstacles [8] (such as modules or logic cells) where the wires of the net must not cross over.

1.2 Scope of Work and Objective

In this study we propose probability-based approaches for rectilinear Steiner tree problems. By considering all possible topologic patterns connecting every pair of pins, we can calculate the probability of the patterns passing over individual edges. The optimal Steiner tree under statistical sense is the tree with maximum sum of the probabilities for all edges of which the tree is comprised. We combine the probabilistic model with other algorithms to improve their performance (i.e., the time complexity and wire length which are major concerns in Steiner tree designs). We also develop a non-deterministic algorithm based on probabilistic model to provide the trade-off between the running speed and the quality of results. Experiment shows that the obtained tree topology is very close to the optimal RST. For large-size problems, the probability-based algorithms are significantly faster than the state-of-the-art algorithm with a slight loss of quality. For small-size problems where the running time is not a big issue, the better solutions can be obtained using our non-deterministic algorithm at the cost of increased running time.

More important, it would be straightforward to extend probabilistic approaches to deal with more general Steiner tree problems with the obstacles or blockages [8]. An optimization algorithm will also be presented, which improves wire length from the results of probabilistic algorithms as well as any other Steiner tree algorithms.

1.3 Orgnization of the Paper

In Chapter 2 minimum spanning tree and algorithms are introduced. Also, Steiner tree and important results are provided. In Chapter 3 some background together with the probabilistic model are described. The Steiner tree construction algorithm and other probability-based algorithms are presented. Also the blockage and congestion problem are discussed. In Chapter 4 data structures and some issues in implement are provided. In Chapter 5 optimization algorithm is presented. In Chapter 6 the results from extensive experiments are given, and comparison with previous works is emphasized in terms of

both wire length and CPU time. In Chapter 7 conclusion and suggestion are given.

Chapter 2

Minimum Spanning Tree and Steiner Tree

Minimum spanning trees are restricted to direct connection between the pins of a net, while in the Steiner tree problem intermediate connection points are permitted to be inserted to reduce the cost of the tree. The minimum spanning tree is important in studying the Steiner tree problem. It can be computed quickly and easily, and provides approximate solution to the Steiner tree. The ratio of Steiner tree to minimum spanning tree is usually used to evaluate the efficiency of Steiner tree algorithm.

There are two main metrics for minimum spanning tree and Steiner Tree: rectilinear and Euclidian distance. In this study we are mainly concerned with the rectilinear distance metrics.

2.1 Minimun Spanning Tree

2.1.1 Definition

Minimum spanning tree is the minimum-weight tree that contains all vertices in a weighted graph.

2.1.2 Algorithms

There are two classic algorithms for minimum spanning tree problems.

(a).Kruskal's Algorithm

This algorithm creates a *forest* of trees. Initially the forest consists of **n** single-node trees (and no edges). At each step, the cheapest edge is added so that it joins two trees together (forming a new tree). If it were to form a cycle, it would simply link two nodes that are already part of a single connected tree. This edge would not be needed, until only a single tree which connects all nodes remains.

(b)Prim's Algorithm

Prim's algorithm is very similar to Kruskal's algorithm. It starts with an arbitrary node as the root of a single tree, and grows a single tree by iteratively adding a node to it using the lowest cost edge, until no unconnected node remains. Both algorithms use the greedy approach – adding the cheapest edge that will not cause a cycle. For a problem with n nodes, both algorithms can be run in O (n log n) time by constructing a Voronoi diagram[18].

2.2 Steiner Tree

The *rectilinear Steiner tree* (RST) problem is defined as follows: given a set P of points in plane, find a set, S, of additional points (called *Steiner points*) such that the length of a minimum spanning tree of P∪S is minimized.

2.2.1 Important Results

Some previous results are important to this study, such as Hanan Grid and Huang's Theorem.

2.2.1.1 Hanan Grid

The Hanan grid is a fundamental structural result to RST problem, which was proposed by Hanan [6] in 1966. Given a set of terminal V, draw horizontal and vertical lines through every terminal (Fig 3.1), the obtained grid is called the Hanan grid for V, and there exists an RST for V that is contained in the Hanan grid. Alternatively, we say that the intersection points in Hanan grid are the candidates of Steiner points.

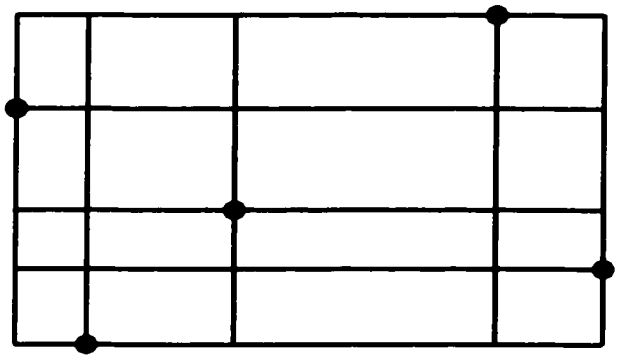


Fig 2.1 Hanan grid

2.2.1.2 Huang's Theorem

For a given terminal set V, let |RST(V)| and |MST(V)| denote the length of an RST and MST for V, respectively. Hwang [9] gave an interesting result about the ratio between |MST| and |RST|, which is no worse than 3/2. This gives us a firm bound on the quality of heuristics that are based on computing MSTs.

2.2.2 Heuristics

Since almost all of heuristics use *rectilinear minimum spanning tree* (MST) as a starting points, we will introduced two typical heuristics for RST: MST embedding and 1-Steiner Heuristics. A comprehensive review of RST heuristics can be found in [10].

2.2.2.1 MST Embeddings

MST embeddings heuristic starts from an MST, and improves it by a series of edge merges. For a pair of adjacent edges in a spanning tree, there is a possibility that by merging portions of the two edges, the tree length can be reduced. An example of this is shown in Fig 3.2. There may be more than one ways in which edges can be merged. The selection of edges and the order of their merging are different from heuristics

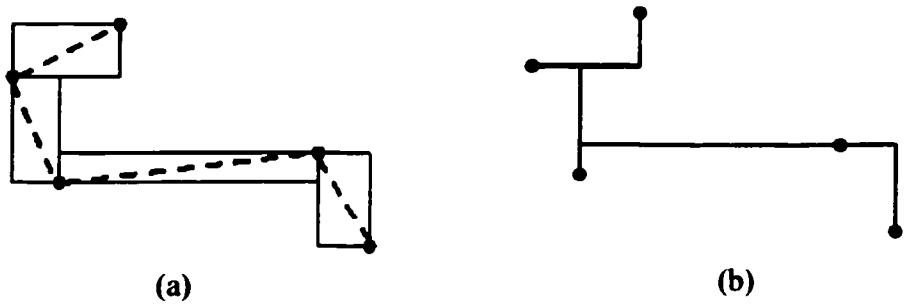


Fig 2.2 MST Embeddings algorithm example

2.2.2.2 1-Steiner Heuristics

This algorithm constructs a Steiner tree through iterative point insertion. At each step, a Steiner point (the best candidate from Hanan grid mentioned in 3.1.1) is added to the point set, until no Steiner point can be found to reduce the MST length. This algorithm is explained in Fig3.2.

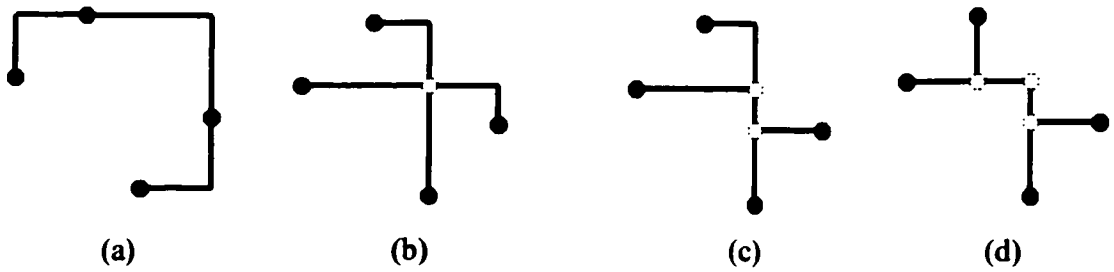


Fig 2.3 1-Steiner heuristic example

Chapter 3

Probabilistic Approaches

In this chapter, Probabilistic model and a series of algorithms based on it are established. Also, how to construct Steiner tree in presence of obstacles is introduced.

3.1 Probabilistic Model

Probabilistic Model includes three parts: Grid Graph, Model and Probability Matrix.

3.1.1 Grid Graph

Consider a set of N points (or pins), $\boldsymbol{P} = \{p_1, p_2, \ldots, p_N\}$ in a plane, where the location of p_i is denoted by (x_i, y_i). Assuming $x_i \neq x_j$ and $y_i \neq y_j$ for $i \neq j$ (More discussions will be given later if this is not the case), we can construct a *grid graph* which consists of the intersections (or, *segments*) of horizontal and vertical lines through all points. It was shown [3] that only those segments within the smallest rectangle enclosing all points need to be considered in obtaining the RST. An optimal RST is a subset of segments, $\boldsymbol{T}$, such that $\boldsymbol{T}$ is a tree for given points and the total wire length over all segments in $\boldsymbol{T}$ is minimum. Fig 3.1 illustrates the grid graph for a set of three points, $\boldsymbol{P} = \{p_1, p_2, p_3\}$. An optimal Steiner tree of Fig3.1 is shown in Fig 3.2, where S_1 is a *Steiner point*.

If we number the columns and rows of the grid graph, the symbol $R(i, j)$ can be used to represent the horizontal segment which lies on row i between columns j and $j + 1$. Similarly, we use $C(i, j)$ to represent the vertical segment which lies on column j between rows i and $i + 1$. For instance, the segments l_1 and l_2 in Fig3.1 are denoted by $R(2, 1)$ and $C(2, 2)$, respectively. Note that the rows are numbered from the bottom to the top in the graph, and the columns are numbered from the left to the right, as shown in Fig 4.1. For

convenience of further discussion, we have the following definitions.

Definition 1: Given two points p_i, $p_j \in \boldsymbol{P}$, the value $m = |c(i) - c(j)|$ is called the *horizontal grid-distance* between them, where $c(i)$ and $c(j)$ are the column numbers of p_i and p_j, respectively. Similarly, the *vertical grid-distance* between them is defined to be $n = |r(i) - r(j)|$, where $r(i)$ and $r(j)$ are the row numbers of p_i and p_j, respectively.

Definition 2: If $0 \le c(j) - c(i) \le 1$ for the two points p_i, $p_j \in \boldsymbol{P}$, then $H(k) = x_j - x_i$ is called the k-th *horizontal physical-length* of the grid graph, where $k = c(i)$. If $0 \le r(j) - r(i) \le 1$, then $V(l) = y_j - y_i$ is called the l-th *vertical physical-length* of the graph, where $l = r(i)$.

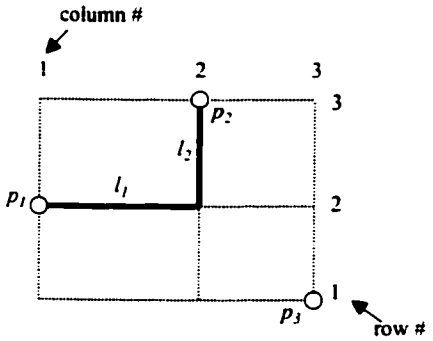


Fig 3.1 The grid graph for a set of three

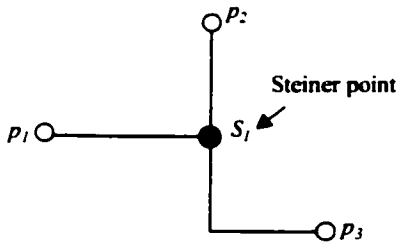


Fig 3.2 An optimal Steiner tree of Fig 3.1

3.1.2 Model

Consider two points p_i, $p_j \in \boldsymbol{P}$ in the grid graph as shown in Fig 3.3. Without loss of generality, we assume $c(i) < c(j)$ and $r(i) > r(j)$. Let $I = r(j)$, and $J = c(j)$. The horizontal and vertical grid-distances between p_i and p_j are $m = c(j) - c(i)$, and $n = r(i) - r(j)$, respectively. Let M be the number of all possible shortest paths from p_i to p_j. The number of those paths which pass through the segment $R(I, J - 1)$ (i.e., R_2 in Fig 3.3) only depends on m and n, and is denoted by $F(m, n)$. The number of those paths which pass through the segment $C(I, J)$ (i.e., C_2 in Fig 3.3) is also a function of m and n, denoted by $G(m, n)$. Obviously, we have $M = F(m, n) + G(m, n)$. In particular, for any positive

integer q, we have $F(1, q) = G(q, 1) = 1$, and $F(q, 1) = G(1, q) = q$.

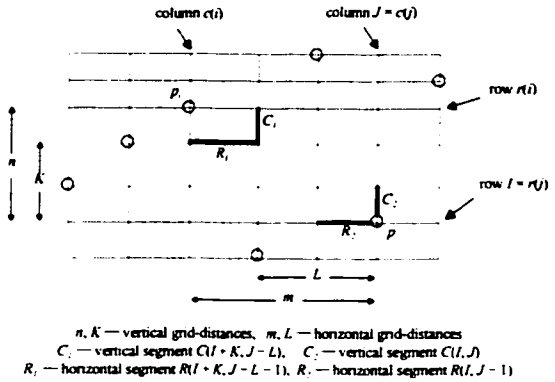


Fig 3.3 Probabilistic analysis for the segments through which the shortest paths between points (p_i and p_j) pass

From a statistical point of view, the probability of a shortest path between the two points passing through $R(I, J-1)$ (or, $C(I, J)$) is given by $F(m, n)/M$ (or, $G(m, n)/M$). From Fig 3.3, $F(m, n)$ and $G(m, n)$ can be written as:

$$F(m,n) = F(m-1,n) + G(m-1,n) \tag{1}$$

and

$$G(m,n) = F(m,n-1) + G(m,n-1) \tag{2}$$

or,

$$F(m,n-1) = F(m-1,n-1) + G(m-1,n-1) \tag{3}$$

and

$$F(m-1,n-1) = G(m-1,n) - G(m-1,n-1) \tag{4}$$

Theorem 1: If we define $F(q, 0) = G(0, q) = 1$ for any integer $q > 0$, then $F(m, n)$ and

$G(m, n)$ can be computed recursively as follows:

$$F(m,n) = \sum_{k=0}^{n} F(m-1,k)$$

$$G(m,n) = \sum_{k=1}^{n} G(m-1,k)$$

and

$$M = F(m+1,n) = G(m,n+1)$$

where $m, n > 1$, and $F(m, n)$, $G(m, n)$, and M are defined as earlier.

The proof of Theorem 1 is omitted. The interested readers are referred to [1].

Furthermore, from Fig 3.3, we can express $F(m, n)$ as follows:

When n = 1,

$$F(m,1) = m \tag{5}$$

When n = 2,

$$F(m,2) = \sum_{k=1}^{m} F(k,1) = \sum_{k=1}^{m} k = \frac{m(m-1)}{2} \tag{6}$$

When n = 3,

$$\begin{aligned} F(m,3) &= \sum_{k=1}^{m} F(k,2) = \sum_{k=1}^{m} \frac{k(k-1)}{2} \\ &= \frac{m^3+3m^2+2m}{6} = \frac{m(m+1)(m+2)}{3\times 2\times 1} \end{aligned} \tag{7}$$

When n = 4,

$$\begin{aligned} F(m,4) &= \sum_{k=1}^{m} F(k,3) = \sum_{k=1}^{m} \frac{k^3+3k^2+2k}{6} \\ &= \frac{m^4+6m^3+11m^2+6m}{24} \\ &= \frac{m(m+1)(m+2)(m+3)}{4\times 3\times 2\times 1} \end{aligned} \tag{8}$$

When n = 5,

$$F(m,5) = \sum_{k=1}^{m} F(k,4) = \sum_{k=1}^{m} \frac{k^4 + 6k^3 + 11k^2 + 6k}{24}$$
$$= \frac{m^4 + 6m^3 + 11m^2 + 6m}{120} \tag{9}$$
$$= \frac{m(m+1)(m+2)(m+3)(m+4)}{5 \times 4 \times 3 \times 2 \times 1}$$

...

More generally, we have

$$F(m,n) = \sum_{k=1}^{m} F(k,n-1) = \frac{m(m+1)(m+2)\cdots(m+n-1)}{n!}$$
$$= \frac{(m+n-1)!}{(m-1)!n!} \tag{10}$$

Similarly, G(m, n) can be expressed as

$$G(m,n) = \frac{(m+n-1)!}{m!(n-1)!} \tag{11}$$

In other words, we have the following theorem:

Theorem 2: The recursive expressions for $F(m, n)$ and $G(m, n)$ are given by

$$F(m,n) = \frac{(m+n-1)!}{(m-1)!n!} = \frac{(m+n-1)}{m-1} \cdot F(m-1,n)$$
$$= \frac{(m+n-1)}{n} F(m,n-1) \tag{12}$$

and

$$G(m,n)=\frac{(m+n-1)!}{m!(n-1)!}=\frac{(m+n-1)}{m}\cdot G(m-1,n)$$
$$=\frac{(m+n-1)}{n-1}G(m,n-1) \tag{13}$$

Proof: According to theorem 1, we need to prove

$$F(m,n)=\sum_{k=0}^{n}F(m-1,k) \tag{14}$$

The right part of equation (14) can be written as

$$\sum_{k=0}^{n}F(m-1,k)=F(m-1,0)+F(m-1,1)+F(m-1,2)+F(m-1,3)+\ldots+F(m-1,n)$$

$$=\frac{(m-2)!}{(m-2)!}+\frac{(m-1)!}{(m-2)!}+\frac{m!}{(m-2)!2!}+\frac{(m+1)!}{(m-2)!3!}+\ldots+\frac{(m+n-2)!}{(m-2)!n!}$$

$$=\frac{(m-2)!}{(m-2)!}+\frac{(m-1)!}{(m-2)!}+\frac{m!}{(m-2)!2!}+\frac{(m+1)!}{(m-2)!3!}+\ldots+\frac{(m+n-2)!}{(m-2)!n!}$$

$$=1+(m-1)+\frac{m!}{(m-2)!2!}+\frac{(m+1)!}{(m-2)!3!}+\ldots+\frac{(m+n-2)!}{(m-2)!n!}$$

$$=\frac{m!}{(m-1)!1!}+\frac{m!}{(m-2)!2!}+\frac{(m+1)!}{(m-2)!3!}+\ldots+\frac{(m+n-2)!}{(m-2)!n!}$$

$$=\frac{2m!+m!(m-1)}{(m-1)!2!}+\frac{(m+1)!}{(m-2)!3!}+\ldots+\frac{(m+n-2)!}{(m-2)!n!}$$

$$= \frac{(m+1)!}{(m-1)!2!} + \frac{(m+1)!}{(m-2)!3!} + \ldots + \frac{(m+n-2)!}{(m-2)!n!}$$

$$= \frac{(m+1)!}{(m-1)!2!} + \frac{(m+1)!}{(m-2)!3!} + \ldots + \frac{(m+n-2)!}{(m-2)!n!}$$

$$= \frac{3(m+1)!+(m+1)!(m-1)}{(m-1)!3!} + \ldots + \frac{(m+n-2)!}{(m-2)!n!}$$

$$= \frac{(m+2)!}{(m-1)!3!} + \ldots + \frac{(m+n-2)!}{(m-2)!n!}$$

$= \ldots$

$$= \frac{(m+n-1)!}{(m-1)!n!}$$

□

Similarly, we can prove (13).

The values of $F(m, n)$ and $G(m, n)$ for $m, n \leq 6$ are shown in Table 4.1 and Table 4.2, respectively.

Table 3.1 $F(m, n)$ function

m \ n	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	3	4	5	6	7
3	3	6	10	15	21	28
4	4	10	20	35	56	84
5	5	15	35	70	126	210
6	6	21	56	126	252	462

Table 3.2 $G(m, n)$ function

m \ n	1	2	3	4	5	6
1	1	2	3	4	5	6
2	1	3	6	10	15	21
3	1	4	10	20	35	56
4	1	5	15	35	70	126
5	1	6	21	56	126	252
6	1	7	28	84	210	462

We note that when m and n are greater than some value (i.e.,15), F(m, n) will increase so rapidly with m and n that we can hardly to see the change in the lower part of the graph. Therefore, we calculate log$F(m, n)$ and give the 3D graph of log$F(m, n)$. $G(m, n)$ and log$G(m, n)$ have almost the same 3D graph with $F(m, n)$ and log$F(m, n)$ because of $G(m, n) = F(n, m)$.

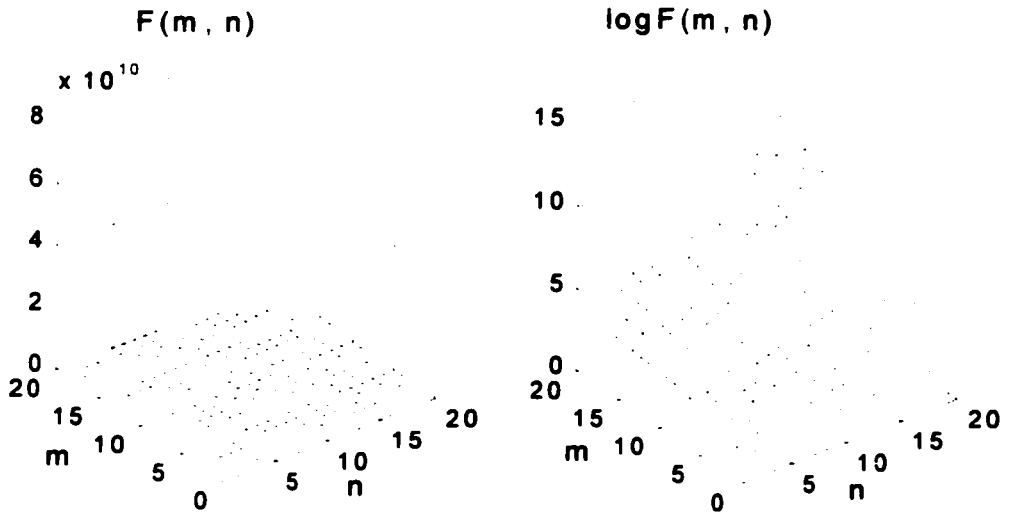


Fig 3.4 The graphic representations for $F(m, n)$ and log $F(m, n)$

3.1.3 Probability Matrix

To calculate the probabilities of segments in Fig 3.3, we take the specific horizontal segment $R(I + k, J - l - 1)$ in the figure, where $0 \le k \le n$, $0 \le l \le m - 1$. Among all M shortest paths from p_i to p_j, the number of paths passing through this segment is given by

$$F(m - l, n - k) \cdot [F(l, k) + G(l, k)] = F(m - l, n - k) \cdot F(l + 1, k)$$

Thus, the probability of a shortest path passing through this segment is

$$PR(I + k, J - l - 1) = \frac{F(m - l, n - k) \cdot F(l + 1, k)}{F(m + 1, n)} \tag{15}$$

Similarly, the probability of a shortest path between p_i and p_j passing through the specific vertical segment $C(I + k, J - l)$ (see Fig 3.3) is given by

$$PC(I + k, J - l) = \frac{G(m - l, n - k) \cdot G(l, k + 1)}{G(m, n + 1)} \tag{16}$$

where $0 \le k \le n - 1$, and $0 \le l \le m$.

If we account for the shortest paths for all $N \cdot (N - 1)/2$ pairs of points in the grid graph, two *probability matrices*, (denoted by ***PR*** and ***PC***), can be used to represent the probabilities of all horizontal and vertical segments, respectively, through which the shortest paths would pass. The element $PR(i, j)$ in ***PR*** (or $PC(i, j)$ in ***PC***) corresponds to the horizontal (or vertical) segment $R(i, j)$ (or $C(i, j)$) in the graph. ***PR*** is an $N \times (N - 1)$ matrix, and ***PC*** is an $(N - 1) \times N$ matrix. The contribution of each pair of points to the matrices is determined by the equations (15) and (16). Intuitively, the greater value of an element implies higher probability that the corresponding segment is to be chosen in obtaining a shortest path (or Steiner tree). An optimal tree under statistical sense is the tree in which the sum of probabilities of the segments involved is maximum.

3.2 Probabilistic Algorithms

Based on the above probabilistic model, we present several algorithms below for Steiner tree construction: *pure probabilistic*, *MST combined*, and *non-deterministic* algorithms. The basic idea of the pure probabilistic algorithm is to select, step by step, the segments with higher probability from the probability matrices until a Steiner tree is constructed. As its name implies, the MST combined algorithm is to join the probabilistic model with Minimal-Spanning-Tree (MST) and only consider the $(N - 1)$ MST edges (compared to $N(N - 1)/2$ pairs of points in the pure probabilistic algorithm) for calculating the probability matrices. The segment selection for Steiner tree construction is the same as in the pure probabilistic algorithm, i.e., choosing the segments one by one in decreasing order of their corresponding probabilities. Differing from these two heuristics that are deterministic, non-deterministic algorithm constructs the tree more "randomly" by generating a group of segments whose appearing probabilities are equal to the corresponding values in the probability matrices. For a given problem, the results may vary each time the algorithm is run on it. The more time the algorithm is executed, the higher probability there is to get the optimal result.

3.2.1 Pure Probabilistic Algorithm

The pseudo-code of pure probability algorithm is given below.

1. *Compute r(i), c(i), H(k), and V(k)*

   Given $\mathbf{P} = \{p_1, p_2, \ldots, p_N\}$, compute the row number r(i) and column number c(i) for p_i, i = 1, 2, ..., N, and compute the horizontal and vertical physical-lengths, i.e., H(k) and V(k) for k = 1, 2, ..., N – 1;

2. *Compute F(m, n) and G(n, m)*

   Compute F(m, n) and G(n, m) for m = 1, 2, ..., N, and n = 0, 1, ..., N – 1;

3. *Obtain PR and PC*

   Obtain the probability matrices, **PR** and **PC**, using equations (15) and (16) for all N· (N – 1)/2 pairs of points, and normalize all elements of **PR** and **PC** (divided by corresponding physical-lengths H(k) or V(k).

4. *Get all points in P connected*

   Select the vertical and horizontal segments one by one in the decreasing order of their corresponding probabilities in **PR** and **PC**. Ignore a specific segment if selecting it would lead to a cycle, until all points in **P** have been connected (using Union and Find data structures).

5. *Delete degree-one-segments*

   Delete degree-one-segments that are not connected to any point in **P**.

6. *Obtain Steiner Points S and calculate wire length*

   Obtain Steiner Points **S**, which is the set of degree-tree and degree-four-points(except those in **P**), and calculate the total wire length of the obtained Steiner tree by calculating the sum physical lengths of selected segments.

Step 1 and Step2 are self-explanatory.

In Step 3 the matrix normalization is necessary since the segments with same probability need to be treated differently, depending on their physical lengths. The shorter segment is selected first in the tree construction so that the total wire length can be reduced. If there are points with the same X-coordinate, i.e., $H(j) = 0$, which implies that no horizontal segments in the j-th column are required, then we delete the j-th column of ***PR*** (instead of dividing it by $H(j)$ as shown in Step 3). Similarly, the i-th row of ***PC*** will be deleted if some points have the same Y-coordinate, i.e., $V(i) = 0$, meaning that no vertical segments in the i-th row are needed.

In Step 4 Union and Find data structures are used for loop detection (see Chapter 4). The

segments that would lead to a circle are ignored in this step, but they are useful in other algorithm (Chapter 5) for improving the tree length because they usually have high probabilities. The time complexity of the above algorithm is $O(N^4)$ due to Step 3 which is computationally expensive. To improve the efficiency, we introduce the MST combined algorithm below.

3.2.2 MST Combined Algorithm

The MST combined algorithm modifies Steps 2 and 3 in the pure probabilistic algorithm. First, it uses *Kruskal*'s algorithm [12] to construct the minimal spanning tree for ***P*** and obtain a set of $(N-1)$ edges, ***MST***. Then, the maximal vertical grid-distance N_R and maximal horizontal grid-distance N_C are calculated, where N_R and N_C are the maximum of $|r(i) - r(j)|$ and $|c(i) - c(j)|$, respectively, for any edge $<i, j> \in$ ***MST***. This is followed by computation of $F(m, n)$ and $G(m, n)$ for $m = 1, 2, \ldots, N_C$ and $n = 1, 2, \ldots, N_R$. Finally, the probability matrices, ***PR*** and ***PC***, are obtained using equations (15) and (16) for all $(N-1)$ pairs of points in ***MST*** and performing the matrix normalization as shown before. The rest of the algorithm is the same as in the pure probabilistic algorithm. It should be noted that while the MST construction requires $O(N \log N)$ time, only $(N-1)$ pairs of points need to be considered. This reduces the time complexity from $O(N^4)$ to $O(N^3)$. In addition, our experiments show that N_R and N_C are normally less than 10, indicating an efficient computation of $F(m, n)$ and $G(m, n)$ even for large design problems. Therefore, the MST combined algorithm is much faster than the pure probabilistic algorithm.

3.2.3 Non-Deterministic Algorithm

The non-deterministic algorithm only modifies the segment selection rule, i.e., Step 4 in the pure probabilistic algorithm, by generating the segments based on their probability values. Assuming that the sum of all entries in the two probability matrices (i.e., ***PR*** and ***PC***) is E, the probability that a segment is to be generated each time is equal to its

corresponding value in the matrix divided by E. The segments with higher values in the matrices are more likely to be chosen from a probabilistic point of view. The other steps involved are the same as in the pure probabilistic algorithm. Once a Steiner tree is constructed, we say that one pass is completed. The wire length for each pass is recorded and compared with the best result from the previous passes. The best result is updated if a shorter wire length is achieved. This process repeats for a given number of passes. Therefore, the algorithm provides the tradeoff between the running time and result quality.

Non-deterministic algorithm:

1 to *3* is same as pure probabilistic algorithm.

*4. Sort **PR** and **PC** in decreasing order, and obtain two arrays **AR** and **AC**, in which we have N·(N-1) elements, respectively, for each element in **AR**, i.e., **AR[i]**, we obtain*

$$SR[i] = \sum_{k=i}^{N\cdot(N-1)} AR[k]$$

*So array **SR** has the same length with **AR**. Similarly we get **SC** whose elements are*

$$SC[i] = \sum_{k=i}^{N\cdot(N-1)} AC[k]$$

*We note that there are not any elements (except 0) same in **SR** or in **SC**. This is very important to later step.*

*5. Multiply **SR** and **SC** by 10 repeatedly until the minimum nonzero elements in **SR** and **SC** are greater than or equal to 1, respectively.*

*6. Obtain the integer part of the fist element (certainly the maximum) of **SR**, we call the integer **ISR**, similarly, we can obtain **ISC**, which is the integer part of the fist element of **SC**.*

7. Generate a random integer number between 0 and ***ISR****, we call* ***RN****, we find the location of* ***RN*** *in* ***SR****, i.e.,* ***SR[l] ≥ RN >SR[l +1]*** *(according to the result form 4 ,* ***SR[l] ≥ RN ≥SR[l +1]*** *is unlikely to occur), where* ***N· (N-1)≥ l≥ 1****, so we can obtain* ***AR[l]*** *and the corresponding probability value in* ***PR****, finally we select corresponding horizontal segment. Similarly, we generate random number* ***CN*** *between 0 and* ***ISC****, and finally select vertical segment, if selecting it would lead to a cycle, ignore the current segment, repeat 7 until all points in* ***P*** *have been connected (using Union and Find data structures).*

8. Delete degree-one-segments is same as pure probability algorithm.

9. Obtain Steiner Points S and calculate wire length is same as pure probability algorithm

10. Let ***Minimum = wire length,*** *and* ***Steiner Points = S.***

11. Do 7, ***8, 9,*** *if* ***wire length < Minimum****, then let* ***Minimum = wire length,*** *and* ***Steiner Points = S,*** *decreasing execution times* ***E*** *(i.e. we initialize* ***E*** *= 10 in our implementation), repeat 11 until* ***E*** *= 0.*

12. We can get wire length from ***Minimum,*** *and obtain Steiner points from* ***Steiner Points.***

3.3 Blockage and Congestion

When we study the rectilinear Steiner tree problem, the main goal is to minimize wire length. In actual global routing, other two problems have to be considered simultaneously: avoidance of "critical regions" and minimizing the wire density. Critical regions are obstacles that the wires of the net must not intersect, such as logic cells. In this paper these obstacles are called "blockages". This problem has been well studied for the case of two-terminal nets. The maze routing technique [19, 20] leads to optimal result for two-terminal nets. But for multi-terminal nets, routing one wire at a time can be difficult because each wire is run without consideration for other terminals. Without this

global consideration, total result is far from optimal. The problem of optimally routing a multi-terminal net in the presence of obstacles has received little attention [21].

In global routing, usually more than one nets (some time over thousands nets) need to be routed in a region. Therefore, wires will be crowded somewhere. This problem is called "congestion". Traffic through the region must be minimized because high wire density will cause serious problem. Previous approaches seem to be difficult to achieve both minimizing wire length and minimizing traffic through the region. In some approaches, a Steiner tree is obtained [22], [23], [24], where length is a purely geometrical concept and traffic has been heuristically minimized as a second object. In other approaches, traffic through the regions has been minimized and heuristics to minimize lengths have been introduced [25], [26], [27].

Probabilistic approaches can achieve the three objects simultaneously, minimizing wire length, minimizing traffic through the region and crossing obstacles, which were attempted by previous approaches but were never truly accomplished. By changing probabilities of edges in some regions, probabilistic approaches can change tree structure in which no edges go through the regions or the number of edges is reduced. Typically the shapes of blockages and congestion are rectangles or combination of rectangles. Therefore, we can define each blockage or congestion by two opposite corner points (the pair of top-left and bottom-right or the pair of top-right and bottom-left). In addition, a value ranged from '0' to '100%' is defined to denote the nature of the area. '0' denotes blockage, in which no edge is allowed to pass. '100%' denotes that the area is not crowded. The value between '0' and '100%' means that the area is congestion area. We note that "blockage" is special case of "congestion'. In our implement, what we need to do is to multiply the probabilities of proper edges by the value.

Chapter 4

Software Design and Implementation

As mentioned in Chapter 3.2, the key step in algorithms is to get the points in **P** connected. The steps are:

1. The forest is constructed with each node in a separate tree.
2. The horizontal and vertical segments are sorted in the decreasing order of their corresponding probabilities in **PR** and **PC**, respectively.
3. Until all nodes in **P** have been connected (in one tree),
    a. Select the vertical and horizontal segments one by one in the sorted edge lists,
    b. If it forms a cycle, reject it,
    c. Else add it to the forest. Adding it to the forest will join two trees together.

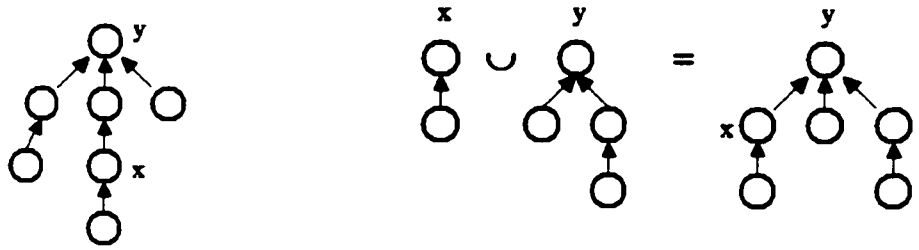
The trick here is to detect cycles and join two trees. For this, we select a *union-find* structure in implementation.

4.1 Union-find Data Structure

Union-find data structure is a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by *union* and *find*.

find(x): start at x and follow pointers up to the root, and return the root (Fig 4.1(a)).

union(x, y): assume x, y are roots of trees, make x a child of y and return y (Fig 4.1(b)).



(a) Find the root of x (b) Merge two tree x and y

Fig 4.1 Find and union examples

Initially all the sub-tree have exactly one node in them. As the algorithm progresses, we form a union of two of the trees, for each sub-tree, we denote one node as the root of that sub-tree. The nominated root somehow, represents each node in the sub-tree.

As we add edge to a tree, we arrange that two nodes of the edges point to their root. As we form a union of two set-trees, we simply arrange that the root of one of the sub-trees now points to any one of the elements of the other sub-tree.

4.2 Loop Detection

Based on *union find* data structure test for a cycle reduces to: for the two nodes at the ends of the candidate edge, find their roots. If the two roots are the same, the two nodes are already in a connected tree and adding this edge would form a cycle. The search for the root simply follows a chain of links.

Each node will need a representative pointer. Initially, each node is its own representative, and the pointer is set to NULL. As the initial pairs of nodes are joined to form a tree, the representative pointer of one of the nodes is made to point to the other,

which becomes the representative of the tree. As trees are joined, the representative pointer of the representative of one of them is set to point to any element of the other. (Obviously, representative searches will be somewhat faster if one of the representatives is made to point directly to the other.)

Each node knows its parent and child and has a rank associated with it. The parent node is always the root node of the set tree. A node's rank is essentially the height of the sub-tree rooted by that node. When the union of two trees is formed, the root with the smaller rank is made to point to the root with the larger rank. Therefore, the height of the tree has been minimized and time has been readuced in finding the root.

4.3 Delete Redundant Edges

It is necessary to delete the useless segments after points in **P** have been connected. Redundant segments are degree-one-segments, in which at least one point neither connects to other segments nor belongs to **P**. Note that deleting degree-one-segments will generate new degree-one-segments. This makes the deleting program iterative.

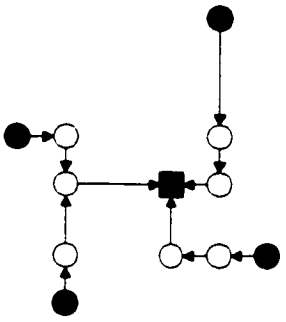
Chapter 5

Optimization Algorithm

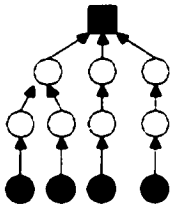
Optimization algorithm starts with probabilistic algorithm result as well as any other algorithm results. It is edge-replacement operation, which introduces a new edge based on probabilities to the existing tree. If a loop is formed, then the longest edge or edge group is removed to break the loop (how to get edge group will be discuss later). Therefore, the wire length is reduced. As the input of optimization algorithm, a directed tree structure with one root is needed. Also, detecting loop and finding the path of loop is the key of the optimization algorithm.

5.1 Directed Tree

In order to detect loop and find the path of loop, a directed tree structure is required. Fig 5.1(a) is an example of directed tree, and a bottom-up form of (a) is shown in (b), a clear way to see the tree structure. Dots denote the original nodes and circles denote other points in grid graph, while square denotes root of the tree, which is essential to the tree and can'. Since deleting root will destroy the tree, the root is selected from original node. In the directed tree, any node can find its parent (node's parent is itself) and each parent has at most four children (left, right, up and bottom).



(a)



(b)

Fig 5.1 Directed tree example

When we add an edge in the tree construction, if two nodes of the edge have the same root (adding this edge will form loop), reject the edge. Otherwise, merge the two trees to which the two nodes belong. There are three cases in merging two trees:

Case1: Only one root is original node. This root will be the root of merged tree.

Case2: Both roots are original nodes.

Case3: Neither of root are original nodes.

For case2 and case 3, we can randomly select one of the two as root of merged tree. Fig 5.2 shows the process to merge two trees. In Fig 5.2 (a), we have two trees whose roots are r_1 and r_2, respectively. If edge e is inserted, the two trees will merge at P_1 P_2.

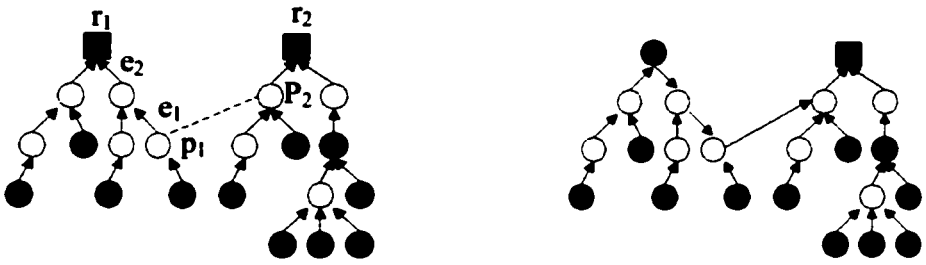


Fig 5.2 Merging two trees

This is Case 2 mentioned above. Assume we select r_2 as the root of merged tree. The merge process is follows:

1. Find the path from p_1 to root and change the direction of the edges (e_1 and e_2) in the path.
2. Edge e point to p_2.

Obviously, r_1 is not a root any more. When all original points have the same root, the directed tree construction is finished. Next step of optimization algorithm is loop detection.

5.2 Loop Detection

During loop detection, we have trace back from the two nodes of added edge to root separately. A loop is formed with the first node they meet. So the longest edge group is deleted to reduce the wire length.

5.3 Edge Replacement

Consider tree fragment for five nodes in Fig 5.3(a). If e_1 and e_2 are inserted to the tree, they form a loop. Suppose e_3, e_4, and e_5 (although e_4 and e_5 look like one segment, they are two different segments in grid graph) are the longest group of edges in the loop. We modify the tree by removing e_3, e_4, and e_5 (Figure 5.3(b)). Therefore, the reduction in the cost of the tree is $(e_3 + e_4 + e_5) - (e_1 + e_2)$. Because e_2 is equal to e_5, and e_3 is equal to e_3, the reduction of cost is the length of e_4.

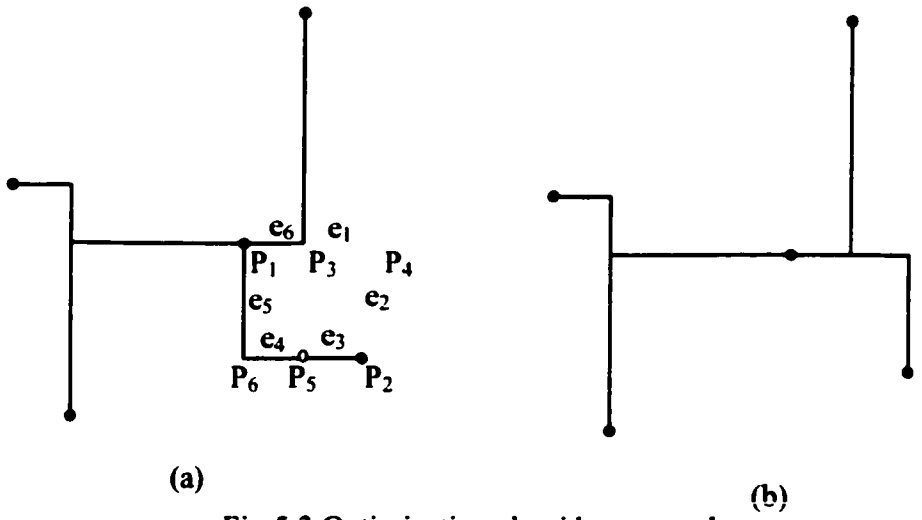


Fig 5.3 Optimization algorithm example

The above procedure replaces three existing edges with two new edges. These groups are called *removable edge groups* (in some case only one edge in the group), which can be deleted together like one edge without destroying the tree. Obviously in each removable edge group (except one edge case) the points connected edges are removable, which are usually degree two points, i.e. P_5 and P_6 in Fig 5.3(a). The two terminals (P_1 and P_2) of

the edges group are *un-removable points*, which are degree three points, degree four points, or original nodes. So a loop is divided into several removable edge groups by un-removable points. In Fig 5.3 (a), edge e_1 to e_6 form a loop, three un-removable points P_1, P_2, and P_3 divide the loop into three removable groups, e_1 e_2, e_3 e_4 e_5, and e_6. The length of edge group e_3 e_4 e_5 is the longest one. Obviously, this group is deleted to reduce the wire length. We note that the degree of some points has been changed during the edge replacement, while some points has been deleted or added with edges. In Fig 5.3(a), before e_1 and e_2 are inserted, P_1 is degree three point (also original node) and P_3 is degree two point. After edge replacement, in Figure 5.3(b), P_1 is degree two point and P_3 is degree three point. Therefore, the degrees of points need to be dynamicly maintaining in implement.

Chapter 6

Experiment and Discussion

We implemented the proposed algorithms and carried out the experiments with Steiner tree construction for problems of various sizes. In order to evaluate the performance, we also implemented the BI1S algorithm [17] which is one of the heuristics with good performance. The programs were run on a *SunBlade* 1000 workstation.

6.1 Small-size Examples

As the first set of experiments, Fig 6.1 through Fig 6.3 show the results for several simple examples using the pure probabilistic algorithm presented in Chapter 3. While the optimal RST is unknown in general, the effectiveness of the algorithm can still be evidenced by inspection of these small-size problems. Particularly, for the case of Fig 6.3 which was taken from [5], the result due to the algorithm of [5] is shown in Fig 6.3(a) with the total wire length of 32, compared to the length of only 30 by our algorithm as shown in Fig 6.3(b). In the following, we use Fig 6.3 to demonstrate the procedure of pure probabilistic algorithm. From Step 1 of the algorithm, we obtain four vectors that denote the row number, column number, horizontal physical-length and vertical physical-length. They are respectively: $\boldsymbol{r}$ = [2 6 5 1 3 4], $\boldsymbol{c}$ = [1 2 3 4 5 6], $\boldsymbol{H}$ = [2 3 1 7 1], and $\boldsymbol{V}$ = [2 2 1 1 5]. From Step 2, both $F(m, n)$ and $G(m, n)$ are calculated, as shown in Tables 3.1 and 3.2. After performing Step 3, we have the following probability matrices:

$$PR = \begin{bmatrix} 0.125 & 0.183 & 1.236 & 0.105 & 0.100 \\ 1.432 & 0.590 & 1.374 & 0.139 & 0.248 \\ 0.469 & 0.391 & 1.786 & 0.388 & 1.038 \\ 0.224 & 0.283 & 1.712 & 0.301 & 3.031 \\ 0.150 & 0.568 & 2.245 & 0.174 & 0.517 \\ 0.100 & 0.651 & 0.648 & 0.036 & 0.067 \end{bmatrix}$$

$$PC = \begin{bmatrix} 0.125 & 0.149 & 0.344 & 1.515 & 0.317 & 0.050 \\ 0.943 & 0.540 & 0.636 & 1.206 & 0.681 & 0.174 \\ 0.948 & 1.231 & 1.788 & 1.610 & 2.038 & 1.386 \\ 0.500 & 1.643 & 2.964 & 1.426 & 0.883 & 0.583 \\ 0.040 & 0.570 & 0.261 & 0.080 & 0.037 & 0.013 \end{bmatrix}$$

Then, Steps 4 through 6 of the algorithm generate the Steiner tree T = {R(2, 1), R(2,2), R(2, 3), R(3, 3), R(3, 4), R(4, 5), R(5, 2), C(1, 4), C(2, 4), C(3, 3), C(3, 5), C(4, 3), C(5, 2)}, as shown in Fig 6.1 (b) which turns out to be an optimal RST. While the proposed algorithm produces the promising results, it is generally not optimal.

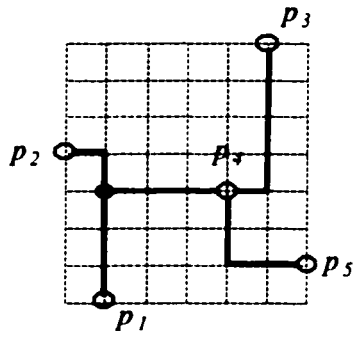


Fig 6.1 The result on an example with N = 5 using pure probabilistic algorithm

In Fig 6.2, for instance, a better solution could be found by replacing the segments R_1 and C_1 with the segment R_2 (shown as the dotted line). For comparison, the experimental data given by different algorithms are shown in Table 6.1, where WL represents the wire length, CPU represents the CPU time (in seconds), and MST corresponds to the *Minimal-Spanning-Tree* resulting from [12]. From this table, it can be seen that the performance of the proposed algorithms is comparable to the BI1S algorithm in terms of wire length, while the non-deterministic algorithm is the slowest.

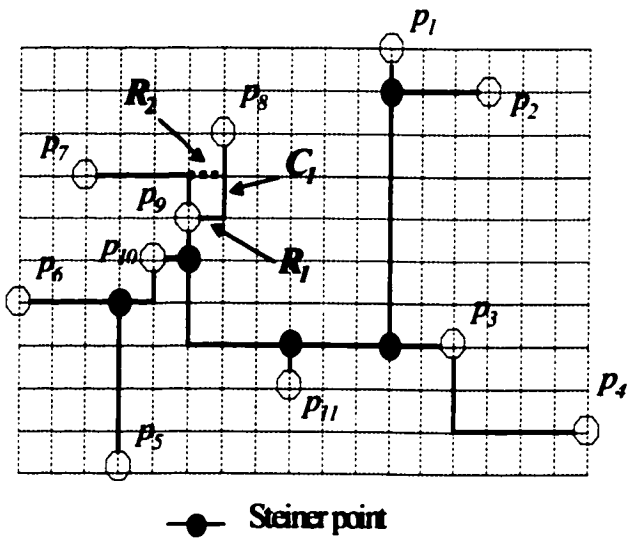
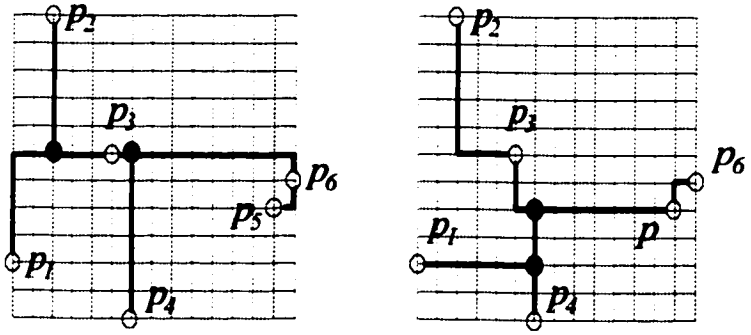


Fig 6.2 The result on an example with N = 11 using pure probabilistic algorithm (Note that replacing R_1 and C_1 with R_2 can lead to a better result).



(a) total wire length: 32 (b) total wire length: 30

Fig 6.3 Comparison of the results by (a) the algorithm from [5] and (b) pure probabilistic algorithm on an example with N = 6.

6.2 Random Examples

As the second set of experiments, we used a large number of test instances that were generated by randomly selecting the coordinates for a set of points from integers ranging between 0 and 10,000. The instance sizes (i.e., the number of points or terminals) are chosen from 3 to 200. The instances are divided into two groups: *group 1* in which the instance sizes are less than 20, and *group 2* in which the instance sizes range from 20 to 200 with increments of ten. For each instance size, we generated 30 instances randomly. The result quality from the proposed algorithms is evaluated using the ratio of RST wire length to MST wire length. The execution time is computed for comparison based on average CPU time needed for each size. For group 1, we tested four algorithms: pure probabilistic, MST combined, non-deterministic, and BI1S algorithms. The results are shown in Fig 6.4 (for wire length comparison) and Fig 6.5 (for CPU time comparison). From Fig 6.4, while the pure probabilistic and MST combined algorithms get less improvement over the minimal spanning tree than the BI1S does, the non-deterministic algorithm is comparable to the BI1S in terms of wire length. Since each point on the curves in Fig 6.4 represents an average of the results over 30 instances, it is clear that the non-deterministic algorithm can outperform the BI1S for some instances. As shown in Fig 6.5, the pure probabilistic and MST combined algorithms are faster than the BI1S. However, the CPU time of the non-deterministic algorithm is longer than the BI1S since many Steiner trees (the selected number of passes is ten in this experiment) have to be constructed. Fortunately, running speed is not an issue for small-size instances. In summary, the probability-based algorithms have very good average performance for small-size problems.

For group 2, we tested three algorithms: pure probabilistic, MST combined, and BI1S algorithms. The results are shown in Fig 6.6 and Fig 6.7. Since the non-deterministic algorithm is computationally expensive for large-size problems, its performance is not shown for this experiment. From Fig 6.6, the pure probabilistic algorithm and MST

combined algorithm get an average of 2% and 6% improvement over MST wire length, respectively, compared to about 11% improvement with the BI1S algorithm. This indicates that the performance of probability-based approaches is still not good enough for large problems that are of theoretical interest. Therefore, further work is needed to enhance their performance. However, the MST combined algorithm is still faster than the BI1S, as can be seen in Fig 6.7. These preliminary experiments show that probabilistic approaches are effective to Steiner problems.

TABLE 6.1
COMPARISON OF DIFFERENT ALGORITHMS ON THREE SMALL-SIZE EXAMPLES

		MST	Pure Probabilistic	MST Combined	Non-deterministic	BI1S
Fig. 5	WL	19	17	17	16	16
	CPU	< 0.01	< 0.01	< 0.01	0.01	< 0.01
Fig. 6	WL	50	45	46	44	43
	CPU	< 0.01	< 0.01	< 0.01	0.025	< 0.01
Fig. 7	WL	35	30	32	30	30
	CPU	< 0.01	< 0.01	< 0.01	0.015	< 0.01

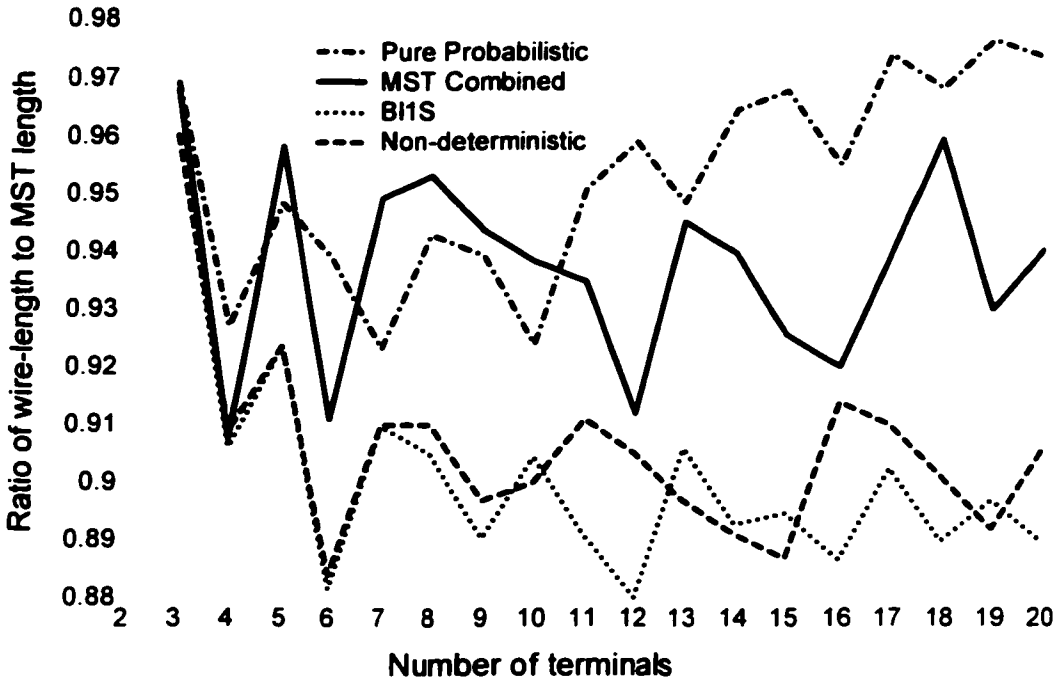


Fig 6.4 Wire length comparison for different algorithms on small-size problems

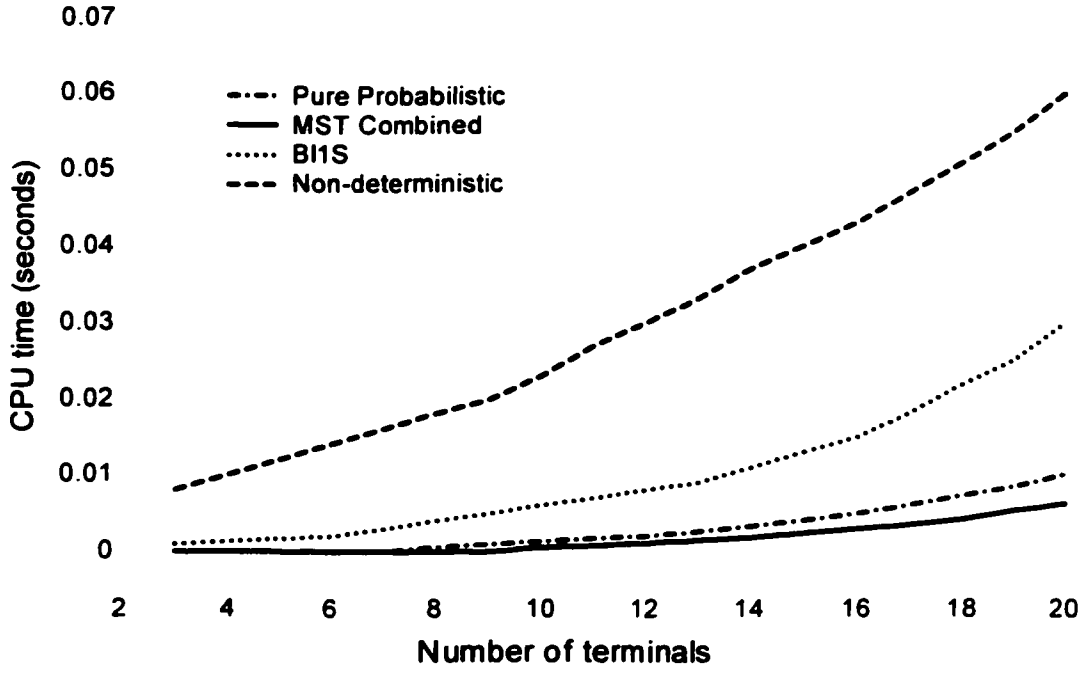


Fig 6.5 Execution time comparison for different algorithms on small-size problems.

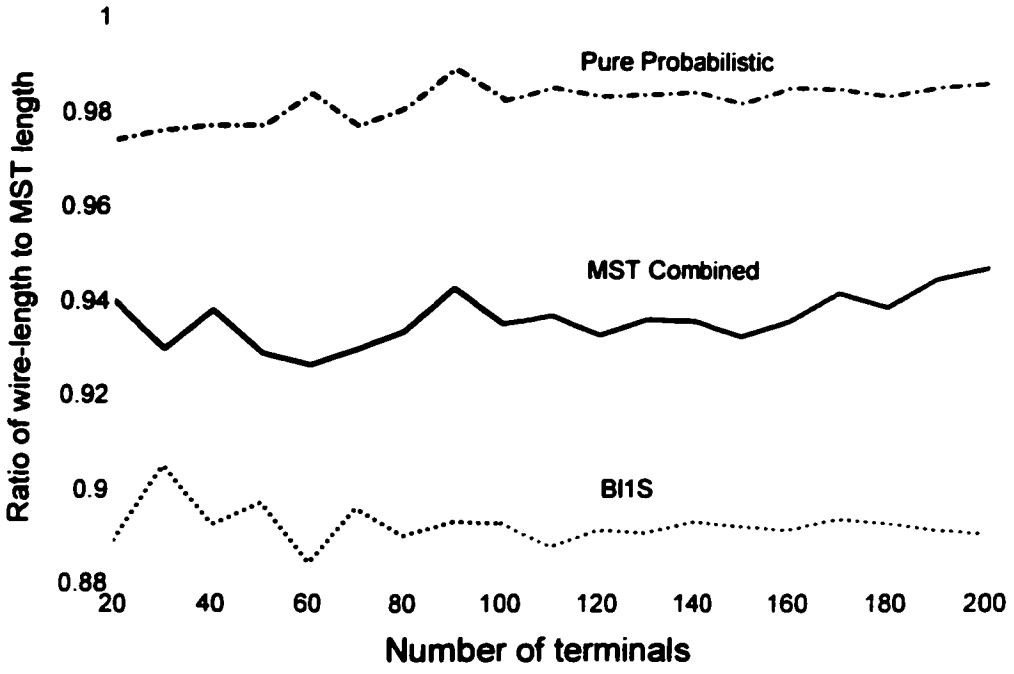


Fig 6.6 Wire length comparison for different algorithms on large-size problems.

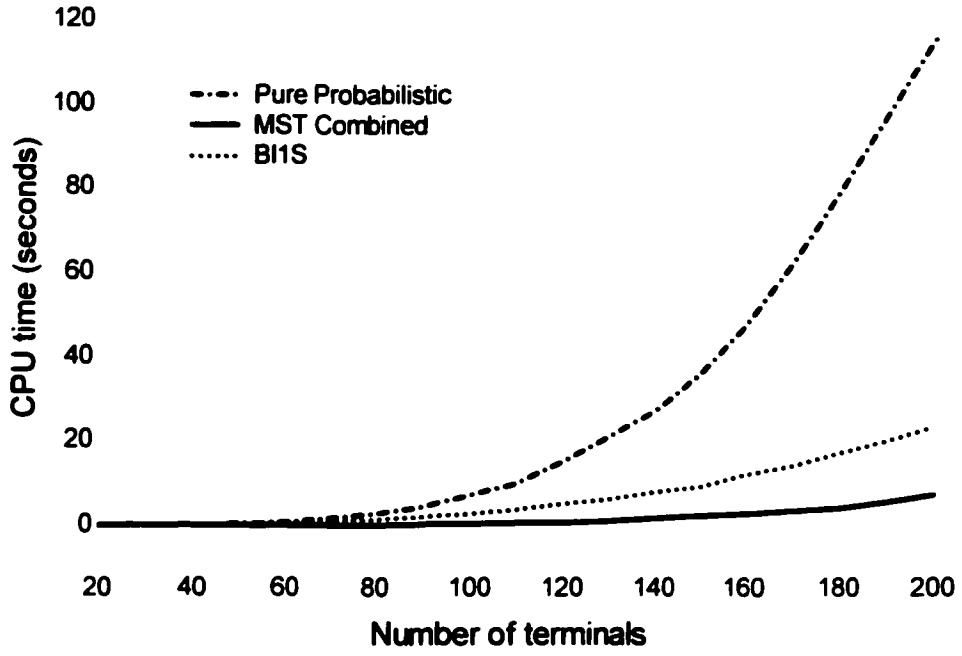


Fig 6.7 Execution time comparison for different algorithms on large-size problems.

6.3 Discussion

There are a few unique potential advantages with the probability-based techniques:

1) The proposed algorithms represent a class of *segment-oriented* methods which construct Steiner trees by selecting segments one after another. The segments with high probability are first selected during the tree construction. In contrast, the BI1S typically belongs to a class of *point-oriented* methods which select (Steiner) points one by one to find the best tree. In this sense, the segment-oriented methods are capable of dealing with congestion-driven interconnect design [8, 9]. This is because the segments falling into the congested areas can be assigned a lower weight so that they would be less likely to be selected in the tree construction. Thus, both wire length and congestion can be minimized by taking into account the wiring flexibility of each individual tree and the effect among different trees.

2) A natural solution for performance/efficiency improvement is to combine the probabilistic algorithm with the BI1S. Intuitively, the points connected to a low-probability segment can be ignored during the tree construction within the BI1S. This speeds up the process without suffering from wire length penalty. Another example for possible improvement is to use *a prior* decomposition/hierarchy technique followed by the probability approach.

3) As mentioned above, calculation of F and G functions in our algorithms is time consuming. Therefore, an important speedup strategy is to obtain their values only once and store them into a look-up table that can be used for different Steiner tree problems, enabling a more efficient computation.

To summarize, our experiments show that the probability-based approaches have desirable performance in terms of computational costs, and generate reasonably good results in terms of wire length minimization. While they are still worse than the state-of-the-art algorithm for large-size problems, we strongly believe that further research can take advantage of the above unique properties to improve the performance.

Chapter 7

Conclusion and Suggestion for Future Work

We have presented a new class of approaches to Steiner tree problems based on probabilistic analysis, with the goal of finding the best solutions under statistical sense. We have described pure probabilistic algorithm, MST combined algorithm and Non-Deterministic algorithm. Also we have designed extensive experients conducted on both small-and large-problems, and it has been shown that probabilistic algorithms have comparable performance with BI1S, the state-of-the-art algorithm. Since the actual VLSI designs have a small number of terminals in each signal net, one can use the non-deterministic algorithm which has the comparable performance to BI1S algorithm in terms of wire length. For large-size problems, the MST combined algorithm is significantly faster than the BI1S algorithm with a slight loss of quality.

More important, we have discussed how to deal with the blockage and congestion problem. An optimization algorithm has been introduced to improve the performance of wire length from the results of probabilitic algorithms as well as other algorithms.

Further research work is needed to improve the algorithms for large-size problems. We also expect to explore the potential advantages of the probabilistic technique and to extend the proposed probability model to other VLSI design applications, such as timing-driven interconnect optimization.

References

[1] C. Chen, J. Zhao, M. Ahmadi, "Probability-Based Approach to Rectilinear Steiner Tree Problems," to appear in *IEEE Trans. VLSI Systems.*

[2] M. R. Gray and D. S. Johnson, "The Rectilinear Steiner Tree Problem is NP-Complete," *SIAM J. Appl. Math*, vol. 32, pp. 826-834, 1977.

[3] M. Hanan, "On Steiner's Problem with Rectilinear Distance," *SIAM J. Appl. Math*, pp. 255-265, 1966.

[4] J. P. Cohoon, D. S. Richards, and J. S. Salowe, "An Optimal Steiner Tree Algorithms for a Net whose Terminals Lie on the Perimeter of a Rectangle," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 4, pp. 398-407, April 1990.

[5] R. Condamoor and I. G. Tollis, "*A New Heuristic for Rectilinear Steiner Trees,*" in *Proceedings of International Symposium on Circuits and Systems*, 1990, pp. 1676-1679.

[6] F. K. Hwang, "On Steiner Minimal Trees with Rectilinear Distance," *SIAM J. Appl. Math*, pp. 104-114, 1976.

[7] F. K Hwang and D. S. Richards, "Steiner Tree Problems," *Networks* 22, pp. 55-89, 1992.

[8] C. J. Alpert, et al, "Steiner Tree Optimization for Buffers, Blockages, and Bays," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 4, pp. 556-562, April 2001.

[9] E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "Creating and Exploiting Flexibility in Steiner Trees," in *Proceedings of 38th IEEE/ACM Design Automation Conference*, June 2001, pp. 195-198.

[10] J. Cong, L. He, C. K. Koh, and P. H. Madden, "Performance Optimization of VLSI Interconnect Layout," *INTEGRATION, the VLSI Journal* 21, pp. 1-94, 1996.

[11] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley, "A New Heuristic for Rectilinear

Steiner Trees," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 10, pp. 1129-1139, October 2000.

[12] J. L. Ganley and J. P. Cohoon, "Routing a Multi-Terminal Critical Net: Steiner Tree Construction in the Presence of Obstacles," in *Proceedings of International Symposium on Circuits and Systems*, pp. 113-116, May 1994.

[13] A. B. Kahng and G. Robins, "A New Class of Iterative Steiner Heuristics with Good Performance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 7, pp. 893-902, July 1992.

[14] M. Borah, R. M. Owens, and M. J. Irwin, "An Edge-Based Heuristic for Steiner Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 12, pp. 1563-1568, December 1994.

[15] D. M. Warme, P. Winter, and M. Zachariasen, "Exact Solutions to Large-Scale Plane Steiner Tree Problems," in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), 1999, pp. 979-980.

[16] D. Z. Du, J. M. Smith, and J. H. Rubinstein, "Advances in Steiner Trees," *Kluwer Academic Publishers*, 2000.

[17] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang, "Closing the Gap: Near-Optimal Steiner Trees in Polynomial Time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 11, pp. 1351-1365, November 1994.

[18] D. T. Lee and C. K. Hong, Voronoi diagrams l1 (l_∞) metrics with 2-dimensional storage applications. *SIAM J. Comput.* 9 (1980) 200-211

[19] C. Y. Lee. An algorithm for path connections and its applications. *IEEE Transactions on Electronic Computers*, 10:346-365, 1961.

[20] E. F. Moore. Shortest path through a maze. *Annals of the Computational Laboratory of Harvard University*, 30:285-292, 1959.

[21] J. L. Ganley and J.P. Cohoon. Routing a Multi-Terminal Critical Net: Steiner Tree Construction in the Presence of Obstacles in *Proceedings of the International Symposium on Circuits and Systems*, pages 113-116, 1994.

[22] M. P. Vecchi and S. Kirkpatrick, "Global wiring by simulated annealing,"*IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 4, pp. 215-222, 1983

[23] A. Aoshima and E.Kuh, "Multi-channel optimization in gate array :SI layout," in *proc. IEEE Int. Conf. Computer-Aided Design*, Nov.1983.

[24] R. Nair, "A simple yet effective technique for global wiring." *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no.2, pp. 165-172, Mar.1987.

[25] J. H. Lee, N. K. Bose, and F. K. Hwang, "Use of steiner's problem in sub-optimal routing in rectilinear metric," *IEEE Trans. Circuits Syst.* vol.CAS-23, pp. 470-476, July 1976.

[26] C. Chiang, M. Sarrafzadeh, and C. K. Wong, "Global routing based on Steiner min-max trees," *IEEE Trans. Computer-Aided Design*, vol. 9. no. 12, pp. 1315-1325,1990.

[27] M. Sarrafzadeh and C. K Wong, " Hierarchical steiner tree construction in uniform orientations," *IEEE Trans. Computer-Aided Design*, vol.11, no. 8, pp. 1095-1103, Sept. 1992.

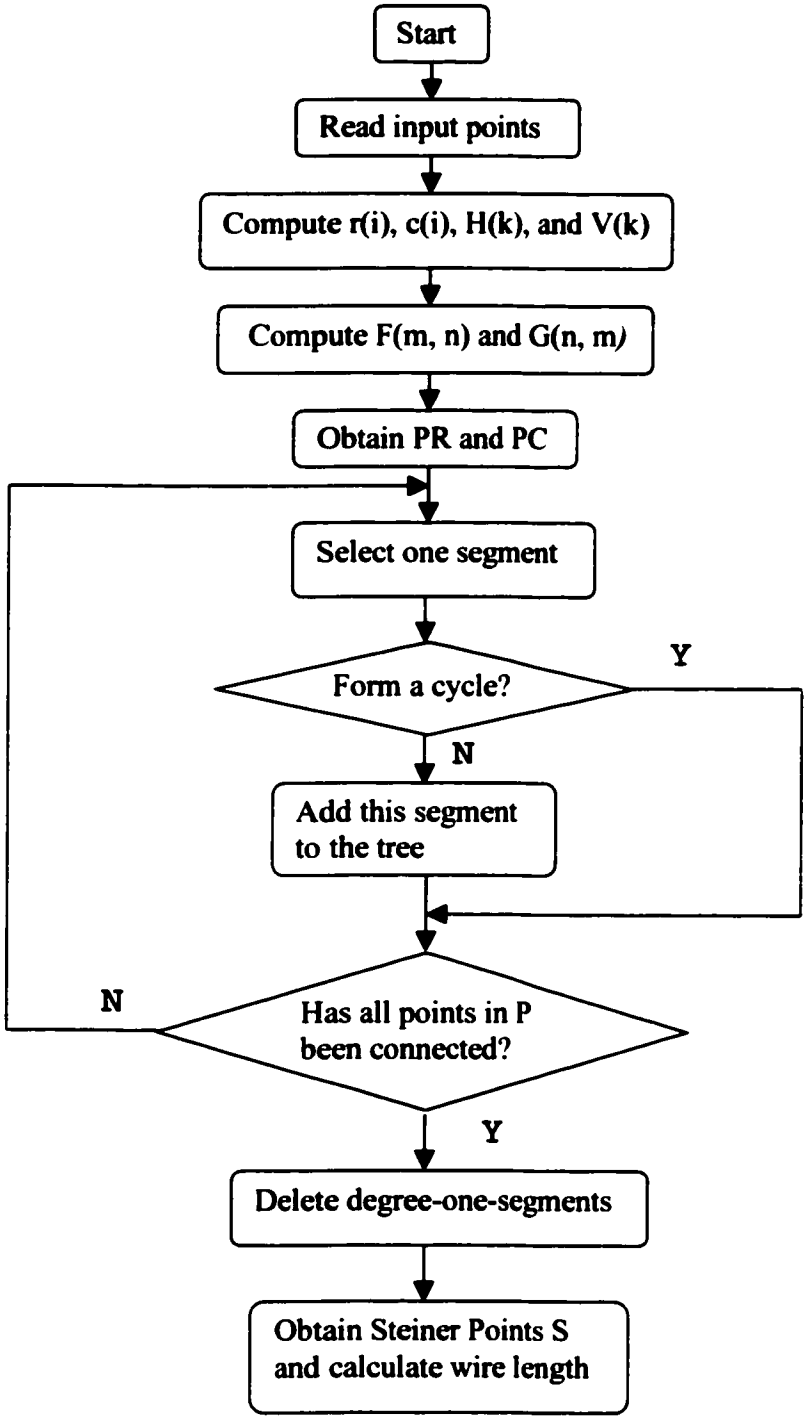
Appendix A

Source Code of Pure Probabilistic Algorithm

This appendix contains Pure Probabilistic source code. It can read coordinates of input points from standard IO, and it displays Steiner tree results on the screen in both graphic format and value format. It also reads inputs from file and writes outputs to file.

Due to space limit, only the main functions are provided here for reference. For the details, please contact the author at zhaog@uwindsor.ca

The flow chart of pure probabilistic algorithm



```
/***********************************
 * Pure probabilistic algorithm  *
 ***********************************/

static void
do_probabilistic(
struct pset *  pts,
struct scale_info *     sip
)
{
   double *        R;
   double *        C;
   double *        X;
   double *        Y;
   double *        F;
   double *        G;
   double *        PR;
   double *        PC;
   double *        PH;
   double *        PV;
   cpu_time_t      T0;
   cpu_time_t      T1;
   cpu_time_t      T2;
   cpu_time_t      T3;
   char            tbuf [20];
   char            tbuf1[20];
   struct point *  p;
   struct point *  p1;
   struct point *  p2;
   struct point *  newp;
   struct edge *   grid;
   struct edge *   ep;
   struct pset *   newpts;
   struct pset *   blockpts;
   int             nedges;
   int             gridp1;
   int             gridp2;
   int             Cgridp1;
   int             Rgridp1;
   int *           Rout;
   int *           Cout;
   int             i;
   int             j;
   int             m;
   int             n;
   int             k;
   int             s;
   int             t;
   int             h;
   char            title [80];
   FILE *          fp;
   FILE *          fpw;
   char *          chars;
```

```
char            ch;
int *           offset;
int             shift;
int             nblocks;
int *           x1;
int *           y1;
int *           x2;
int *           y2;
int             temp;
int             rate;
int             xl, xr, yl, yh;
int             buf[100];
double          wire_len;
double          ratio;

N = pts -> n;



/*****************************************************
 *     step1 Compute r(i), c(i), H(k), and V(k)  *
 *****************************************************/

T0 = get_cpu_time ();

C = NEWA ( N + 1, double);
R = NEWA ( N + 1, double);
for (p = &(pts->a[0]), i=0; i<N; i++, p++){
     C[i] = p->x;
     R[i] = p->y;
}

X = NEWA ( N + 1, double);
Y = NEWA ( N + 1, double);
for (i=0; i<N; i++){
     X[i] = C[i] ;
     Y[i] = R[i] ;
}
#ifdef DISPLAYMATRIX

printf("\n");
printf("%d Define Vertices:\n",N);
for (i=0; i<N; i++) printf("p%d (%f%f)\n",i+1,C[i],R[i]);
#endif
incr_sort(C, N);
incr_sort(R, N);

r = NEWA ( N + 1, int);
c = NEWA ( N + 1, int);
for (i=0; i<N; i++){
    j=0;
    while (X[i]!=C[j]) j++;
    c[i]=j+1;
```

```
}
for (i=0; i<N; i++){
    j=0;
    while (Y[i]!=R[j]) j++;
    r[i]=j+1;
}
#ifdef DISPLAYMATRI
printf("\nr = {");
for (i=0; i<N; i++) printf("%3d",r[i]);
printf(" }\n");
                    printf("c = {");
for (i=0; i<N; i++) printf("%3d",c[i]);
                    printf(" }\n");

                    printf("\nx = {");
for (i=0; i<N; i++) printf("%f",C[i]);
                    printf(" }\n");
                    printf("y = {");
for (i=0; i<N; i++) printf("%f",R[i]);
                    printf(" }\n");
#endif


H = NEWA ( N + 1, double);
V = NEWA ( N + 1, double);
#ifdef DISPLAYMATRIX
    printf("\nH = {");
#endif
for (i=0; i<N-1;i++){
    H[i]=C[i+1]-C[i];
    V[i]=R[i+1]-R[i];
#ifdef DISPLAYMATRIX
    printf("%f",H[i]);
#endif
}
#ifdef DISPLAYMATRIX
    printf(" }\n");
                      printf("V = {");
for (i=0; i<N-1; i++) printf("%f",V[i]);
                      printf(" }\n");
#endif
```

```
/****************************************
 * step2 Compute F(m, n) and G(n, m)  *
 ****************************************/

 F = NEWA ((N+1)*(N+1)+1, double);
 G = NEWA ((N+1)*(N+1)+1, double);

 for (m=0; m<=N; m++)
    for (n=0; n<=N; n++) *(F + (N+1)*m + n) = 0;

 for (m=1; m<=N; m++) *(F + (N+1)*m + 0) = 1;  /* F[m][0] = 1;*/
 for (n=1; n<=N; n++) *(F + (N+1)*1 + n) = 1;  /* F[1][n] = 1;*/

 for (m=1; m<=N; m++)
    for (n=1; n<=N; n++)
          for (k=0; k<=n; k++)
                *(F + (N+1)*m + n) += *(F + (N+1)*(m-1) + k);
                      /*F[m][n]+=F[m-1][k];*/


#ifdef DISPLAYMATRIX
          printf("\nF(m,n) =");
 for (m=1; m<=N; m++){
    printf(" [");
          for (n=1; n<=N; n++)
                printf("%e ", *(F + (N+1)*m + n));
          printf(" ]\n\t");
 }
#endif

 for (m=0; m<=N; m++)
    for (n=0; n<=N; n++) *(G + (N+1)*m + n) = 0;

 for (n=1; n<=N; n++) *(G + (N+1)*0 + n) = 1;  /*G[0][n] = 1;*/
 for (m=1; m<=N; m++)
    for (n=1; n<=N; n++)
          for (k=1; k<=n; k++)
                *(G + (N+1)*m + n) += *(G + (N+1)*(m-1) +k);
                /*G[m][n]+=G[m-1][k];*/
 #ifdef DISPLAYMATRIX
     printf("\nG(m,n) =");
 for (m=1; m<=N; m++){
     printf(" [");
     for (n=1; n<=N; n++)
          printf("%e ", *(G + (N+1)*m + n));
     printf(" ]\n\t");
 }
 printf("\n");
 #endif
```

```
/***************************
 * step3 Obtain PR and PC *
 ***************************/

   PR = NEWA ((N+1)*(N+1)+1, double);
   PC = NEWA ((N+1)*(N+1)+1, double);
   for (m=0; m<=N; m++)
       for (n=0; n<=N; n++) *(PR + (N+1)*m + n) = 0;
       /*PR[m][n] = 0;*/

   for (m=0; m<=N; m++)
        for (n=0; n<=N; n++) *(PC + (N+1)*m + n) = 0;
       /*PC[m][n] = 0;*/

   for (s=0; s<N; s++)
        for (t=0; t<N; t++){
            if(s<t)  PR_PC(s, t, F, G, PR, PC);
        }
   #ifdef DISPLAYMATRIX
        printf("\nPR(i,j)=");
   for (i=1; i<=N; i++){
        printf(" [");
        for (j=1; j<N; j++)
            printf("%2f ",*(PR + (N+1)*i + j));
        printf(" ]\n\t");
   }

   printf("\nPC(i,j)=");
   for (i=1; i<N; i++){
        printf(" [");
        for (j=1; j<=N; j++)
            printf("%7f",*(PC + (N+1)*i + j));
        printf(" ]\n\t");
   }
        printf("\n");

  #endif

   PH = NEWA ((N+1)*(N+1)+1, double);
   PV = NEWA ((N+1)*(N+1)+1, double);
   for (j=1; j<=N; j++)
       for (i=1; i<N; i++){
            *(PH + (N+1)*j + i) = *(PR + (N+1)*j + i)/H[i-1];
            /*sqrt(H[i-1]) */
            /*
             *PH[j][i] = PR[j][i]/ H[i-1];
             */
        }

    for (j=1; j<=N; j++)
       for (i=1; i<N; i++){
         *(PV + (N+1)*i + j) = *(PC + (N+1)*i + j)/V[i-1];
```

```
            /*sqrt(V[i-1])*/
            /*
             *PV[i][j] = PC[i][j]/ V[i-1];
             */
          }

    #ifdef DISPLAYMATRIX
              printf("\nPR / H =");
    for (i=1; i<=N; i++){
         printf(" [");
         for (j=1; j<N; j++)
             printf("%7f ",  *(PH + (N+1)*i + j));
         printf(" ]\n\t");
   }

         printf("\nPC / V =");
    for (i=1; i<N; i++){
         printf(" [");
         for (j=1; j<=N; j++)
             printf("%7f ", *(PV + (N+1)*i + j));
         printf(" ]\n\t");
    }
         printf("\n");

    #endif



/*************************************
 *    Blockages and congestion       *
 *************************************/

    if (Exist_Blockage) {
       printf("exist blockage\n");
       offset = NEWA(2, int);
       chars = NEWA(15, char);
       fp = fopen("blockage.txt", "r");
       read (fp, chars);
       nblocks = decode (chars, ';', 2, offset);
       x1 = NEWA(nblocks, int);
       y1 = NEWA(nblocks, int);
       x2 = NEWA(nblocks, int);
       y2 = NEWA(nblocks, int);
       for (i = 0; i < nblocks; i++){
              ch = getc(fp);
              read (fp, chars);
              x1[i] = decode (chars, ',', 3, offset);
              shift  = offset[0] - 1;
              y1[i] = decode (chars, ')', 5 + shift, offset);

              ch = getc(fp);
              read (fp, chars);
              x2[i] = decode (chars, ',', 3, offset);
```

```
 shift  = offset[0] - 1;
 y2[i] = decode (chars, ')', 5 + shift, offset);

 ch = getc(fp);
 read (fp, chars);
 rate = decode (chars, '%', 5, offset);

 if(x1[i] > x2[i]){
      temp    = x1[i];
      x1[i]   = x2[i];
      x2[i]   = temp;
 }
 if(y1[i] > y2[i]){
      temp    = y1[i];
      y1[i]   = y2[i];
      y2[i]   = temp;
 }
 printf("p1(%d, %d)\n", x1[i], y1[i]);
 printf("p2(%d, %d)\n", x2[i], y2[i]);
 printf("rate = %d\n", rate);

  xl = xr = yl = yh = 0;
    for (j = 0; j<N; j++){
      if (X[j]<  (double)x1[i]) ++xl;
      if (X[j]<= (double)x2[i]) ++xr;
      if (Y[j]<  (double)y1[i]) ++yl;
      if (Y[j]<= (double)y2[i]) ++yh;
}

printf("(%d, %d), (%d, %d)\n", xl, yl, xr, yh);

for (j=yl+1; j<yh+1; j++)
      for (k=xl; k<=xr; k++){
            *(PH + (N+1)*j + k) *= rate;
            *(PH + (N+1)*j + k) /= 100;

            /*
            *PH[j][i] = PR[j][i]/ H[i-1];
            */
}

for (j=xl+1; j<xr+1; j++)
      for (k=yl; k<=yh; k++){

            *(PV + (N+1)*k + j) *= rate;
            *(PV + (N+1)*k + j) /= 100;

            /*
            *PV[i][j] = PC[i][j]/ V[i-1];
            */
      }

}
```

```
            fclose(fp);
            free ((char *) offset);
            free ((char *) chars);
    }


#ifdef DISPLAYMATRIX
   printf("\nPR / H =");
   for (i=1; i<=N; i++){
        printf(" [");
        for (j=1; j<N; j++)
            printf("%7f ",  *(PH + (N+1)*i + j));
        printf(" ]\n\t");
   }

   printf("\nPC / V =");
   for (i=1; i<N; i++){
        printf(" [");
        for (j=1; j<=N; j++)
            printf("%7f ", *(PV + (N+1)*i + j));
        printf(" ]\n\t");
   }
   printf("\n");

#endif


    if (Exist_Blockage) {
      blockpts = NEW_PSET (nblocks * 5);

      blockpts->n = nblocks * 5;

      p = &(blockpts->a[0]);
      for(i = 0; i < nblocks; i++, p++){
            p->pnum = i*5;
            p->x = x1[i];
            p->y = y1[i];
            ++p;
            p->pnum = i*5+1;
            p->x = x1[i];
            p->y = y2[i];
            ++p;
            p->pnum = i*5+2;
            p->x = x2[i];
            p->y = y2[i];
            ++p;
            p->pnum = i*5+3;
            p->x = x2[i];
            p->y = y1[i];
            ++p;
            p->pnum = i*5+4;
            p->x = x1[i];
```

```
        p->y = y1[i];
    }
}

newpts = NEW_PSET (N*N);

newpts->n = N*N;

for (p = &(newpts->a[0]), i=0; i<N*N; i++, p++){
     p->pnum = i;

     gridp1 = i;
     gridp2 = i;
     gridp1 %= N;
     gridp2 /= N;
     p->x = C[gridp1];
     p->y = R[gridp2];
     /*
     printf("p%d (%f, %f)\n", p->pnum, p->x, p->y);
     */
  }

 grid = NEWA(2*N*(N-1), struct edge);
 loop_edges = NEWA(2*N*(N-1), struct edge);
 num_loop_edges = 0;
 tree_edges = NEWA(2*N*(N-1), struct edge);
 redundant_edges = NEWA(2*N*(N-1), struct edge);
 num_redundant_edges = 0;

 nedges = probabilistic(PH, PV, &grid [0]);

 /*
 for (i = 0; i < nedges; i++) {
        ep = &grid[i];
        gridp1 = ep -> p1;
        gridp2 = ep -> p2;
        printf("point = %d parent = %d weight = %f)\n", gridp1,
        Tree[gridp1].parent, Tree[gridp1].weight);
        printf("point = %d parent = %d weight = %f)\n", gridp2,
        Tree[gridp2].parent, Tree[gridp2].weight);
}
*/

 pro_len = 0;
 wire_len = 0;
 for (i = 0; i< nedges; i++) {
      ep = &grid[i];
      gridp1 = ep->p1;
      gridp2 = ep->p2;

      if ((gridp2 - gridp1)>1){
           gridp1 -= 1;
               Cgridp1 = gridp1;
```

```
                Cgridp1 %= N;
                gridp1 /= N;
            pro_len += V[gridp1];
        }else
        {
            gridp1 -= 1;
            Rgridp1 = gridp1;
            Rgridp1 /= N;
            gridp1 %= N;
            pro_len += H[gridp1];
        }
    }

    T1 = get_cpu_time ();

    fpw = fopen("pure.ps", "a");

    (void) printf ("\n  %% probabilistic\n");
    fprintf (fpw, "\n  %% probabilistic\n");

    begin_plot (BIG_PLOT);

    (void) printf ("\tPlot_Terminals\n");
    fprintf (fpw, "\tPlot_Terminals\n");

    for (i = 0; i < nedges; i++) {
        ep = &grid[i];
        p1 = &(newpts -> a [ep -> p1 - 1]);
        p2 = &(newpts -> a [ep -> p2 - 1]);
        draw_segment (p1, p2, sip);
    }

#ifdef DISPLAYREDUNDANT
    for (i = 0; i < num_redundant_edges; i++) {
        ep = &redundant_edges[i];
        p1 = &(newpts -> a [ep -> p1 - 1]);
        p2 = &(newpts -> a [ep -> p2 - 1]);
        draw_segment2 (p1, p2, sip);
    }
#endif

#ifdef DISPLAYLOOPEDGES
    for (i = 0; i < num_loop_edges; i++) {
        ep = &loop_edges[i];
        p1 = &(newpts -> a [ep -> p1 - 1]);
        p2 = &(newpts -> a [ep -> p2 - 1]);
        draw_segment1 (p1, p2, sip);
    }
#endif


    if (Exist_Blockage) {
       for (i = 0; i < nblocks; i++){
```

```
            for (j = 0; j < 4; j++) {
                  p1 = &(blockpts -> a [i*5 + j]);
                  p2 = &(blockpts -> a [i*5 + j+1]);
                  draw_segment1 (p1, p2, sip);
            }
      }
   }

   convert_cpu_time (T1-T0, tbuf);
   (void) sprintf (title, "Probabilistic    length: %f, CPU time:
                  %s seconds\n",   pro_len, tbuf);

   end_plot (title);
   fclose(fpw);


   T2 = get_cpu_time ();

   num_tree_edges = add_looped_edges();

   for (i = 0; i < num_tree_edges; i++) {
         ep = &tree_edges[i];
         gridp1 = ep -> p1;
         gridp2 = ep -> p2;
#ifdef DISPLAYOPTIMIZED
         printf("point = %d parent = %d weight = %f)\n", gridp1,
Tree[gridp1].parent, Tree[gridp1].weight);
         printf("point = %d parent = %d weight = %f)\n", gridp2,
                Tree[gridp2].parent, Tree[gridp2].weight);
#endif
}


   for (i = 0; i< num_tree_edges; i++) {
        ep = &tree_edges[i];
        gridp1 = ep->p1;
        gridp2 = ep->p2;

        if ((gridp2 - gridp1)>1){
             gridp1 -= 1;
             Cgridp1 = gridp1;
             Cgridp1 %= N;
             gridp1 /= N;
             wire_len += V[gridp1];
         }else
         {
             gridp1 -= 1;
             Rgridp1 = gridp1;
             Rgridp1 /= N;
             gridp1 %= N;
             wire_len += H[gridp1];
         }
    }
```

```
    T3 = get_cpu_time ();
    ratio = 100*wire_len/pro_len;

    fpw = fopen("optimized.ps", "a");
    begin_plot (BIG_PLOT);


    (void) printf ("\tPlot_Terminals\n");
    fprintf (fpw, "\tPlot_Terminals\n");


    for (i = 0; i < num_tree_edges; i++) {
          ep = &tree_edges[i];
          p1 = &(newpts -> a [ep -> p1 - 1]);
          p2 = &(newpts -> a [ep -> p2 - 1]);
          draw_segment (p1, p2, sip);
    }


    if (Exist_Blockage) {
    for (i = 0; i < nblocks; i++){
          for (j = 0; j < 4; j++) {
                p1 = &(blockpts -> a [i*5 + j]);
                p2 = &(blockpts -> a [i*5 + j+1]);
                draw_segment1 (p1, p2, sip);
          }
     }
}


    convert_cpu_time (T3-T2, tbuf);
    (void) sprintf (title, "Optimized length: %f, Ratio= %2f%%, CPU
    time: %s seconds\n",   wire_len, ratio, tbuf);

    end_plot (title);

    fclose(fpw);

    if (Exist_Blockage) {
      free ((char *) x1);
      free ((char *) y1);
      free ((char *) x2);
      free ((char *) y2);
      free ((char *) blockpts);
    }
     free ((char* ) degree);
     free ((char *) tree_edges);
     free ((char *) Tree);
     free ((char *) loop_edges);
     free ((char *) grid);
     free ((char *) newpts);
```

```
    free ((char *) PV);
    free ((char *) PH);
    free ((char *) PC);
    free ((char *) PR);
    free ((char *) G);
    free ((char *) F);
    free ((char *) V);
    free ((char *) H);
    free ((char *) c);
    free ((char *) r);
    free ((char *) Y);
    free ((char *) X);
    free ((char *) R);
    free ((char *) C);
}
```

```
/*****************************************
 * step4 Get all points in P connected  *
 *****************************************/

int
probabilistic (

 double  *         PH,
 double  *         PV,
 struct edge *     edges
)
{
int                nedges_row;
int                nedges_col;
int                pro_edge_count;
struct edge *      edge_array_row;
struct edge *      edge_array_col;

      nedges_row = build_pro_edges_row (PH, &edge_array_row, 0);
      nedges_col = build_pro_edges_col (PV, &edge_array_col, 0);

      pro_edge_count = pro_edge_list (nedges_row,nedges_col,
      &edge_array_row [0], &edge_array_col [0],edges);
      free ((char *) edge_array_col);
      free ((char *) edge_array_row);
      return (pro_edge_count);
}

int
pro_edge_list (

int                nedges_row,
int                nedges_col,
struct edge *      edge_list_row,
struct edge *      edge_list_col,
struct edge *      edges
)
{
 int edge_count;

       pro_sort_edge_list (edge_list_row, nedges_row);
       pro_sort_edge_list (edge_list_col, nedges_col);

       edge_count = make_a_tree ( nedges_row,
                                  nedges_col,
                                  edge_list_row,
                                  edge_list_col,
                                  edges );

       pro_len = calculate_len(edge_count, edges);

       return (edge_count);
```

```
/*****************************************
 * make_a_tree                           *
 *****************************************/


int
make_a_tree(

int             nedges_row,
int             nedges_col,
struct edge *   edge_list_row,
struct edge *   edge_list_col,
struct edge *   edges
)
{
 int              i;
 int              max_vert;
 int              pro_edge_count;
 struct dsuf      sets;
 int              random_row;
 int              random_col;
 struct edge *    ep_row;
 struct edge *    ep_col;
 struct edge *    ep1;
 struct edge *    ep2;
 int              root1;
 int              root2;
 int              connected;
 int              root0;
 int              root;

 int *            marked;
 int              point1;
 int              point2;
 int              lp;   /*location of point*/

      Tree   = NEWA(N*N + 1, struct node);
      marked = NEWA(N*N + 1, int);
      for (i = 0; i <= N*N; i++) marked[i] = 0;

      max_vert = 1;
      ep_row = edge_list_row;
      for (i = 0; i < nedges_row; i++, ep_row++) {
            if (ep_row -> p1 > max_vert) {
                  max_vert = ep_row -> p1;
            }
            if (ep_row -> p2 > max_vert) {
                  max_vert = ep_row -> p2;
            }
      }

      dsuf_create_pro (&sets,  max_vert + 1);
      ep_row = edge_list_row;
```

```
for (i = 0; i < nedges_row; i++, ep_row++) {
      dsuf_makeset_pro (&sets, ep_row->p1);
          dsuf_makeset_pro (&sets, ep_row->p2);
}

pro_edge_count = 0;

ep_row = edge_list_row;
ep_col = edge_list_col;
#ifdef DISPLAYEDGES
printf("\nR(i, j)\t\tC(i, j):\n");
#endif
do {
      root1 = dsuf_find_pro (&sets, ep_row -> p1);
      root2 = dsuf_find_pro (&sets, ep_row -> p2);

      if (root1 NE root2) {
                  #ifdef DISPLAYEDGES
            printf("%f\t", ep_row->len);
                  #endif
            dsuf_unite_pro (&sets, root1, root2);
            *edges = *ep_row;
            ++edges;
            ++pro_edge_count;

            point1 = ep_row -> p1;
            point2 = ep_row -> p2;
            make_node(marked, point1, point2, 0);
      }else
      {
                  #ifdef DISPLAYEDGES
            printf("......\t\t");
                  #endif
            *loop_edges = *ep_row;
            ++loop_edges;
            ++num_loop_edges;
      }
      ++ep_row;

      root1 = dsuf_find_pro (&sets, ep_col -> p1);
      root2 = dsuf_find_pro (&sets, ep_col -> p2);
      if (root1 NE root2) {
                  #ifdef DISPLAYEDGES
            printf("%f\n", ep_col->len);
                  #endif
            dsuf_unite_pro (&sets, root1, root2);
            *edges = *ep_col;
            ++edges;
            ++pro_edge_count;

            point1 = ep_col -> p1,
            point2 = ep_col -> p2;
            make_node(marked, point1, point2, 1);
```

```
        }else
        {
                    #ifdef DISPLAYEDGES
              printf("...\n");
                    #endif
              *loop_edges = *ep_col;
              ++loop_edges;
              ++num_loop_edges;

        }
        ++ep_col;

        root0 = dsuf_find_pro(&sets, (r[0]-1)*N+c[0]);
        connected = 0;

        for (i = 1; i<N; i++){
                    root = dsuf_find_pro(&sets, (r[i]-1)*N+c[i]);
                    if (root != root0) {
              connected = 1;
              break;
                   }
          }

}while(connected > 0);

  /*put the rest of edges into loop_edges list*/
  while ((ep_row != edge_list_row+ nedges_row)&&(ep_col !=
  edge_list_col+ nedges_col) ) {
      if (ep_row -> len > 0){
            *loop_edges = *ep_row;
            ++loop_edges;
            ++num_loop_edges;
      }
      ++ep_row;

      if (ep_col -> len > 0){
            *loop_edges = *ep_col;
            ++loop_edges;
            ++num_loop_edges;
      }
      ++ep_col;
}

edges -= pro_edge_count;
loop_edges -= num_loop_edges;

ep1 = edges;
ep2 = tree_edges;
for (i = 0; i < pro_edge_count; i++, ep1++, ep2++) *ep2 =
*ep1;
num_tree_edges = pro_edge_count;
```

```
        degree = NEWA(N*N + 1, int);
        for (i=0; i<=N*N;i++) degree[i]=0;
        pro_edge_count = delete_degree_one(edges, pro_edge_count,
        degree, 1); /*1 -- keep redundant, 0 -- not */
        dsuf_destroy_pro (&sets);

        free((char*) marked);

        return(pro_edge_count);
}
```

```
/**********************************
 *     calculate tree length      *
 **********************************/


 calculate_len(

 int             nedges,
 struct edge *   grid
 )
 {
  dist_t         length;
  int            i;
  int            gridp1;
  int            gridp2;
  int            Cgridp1;
  int            Rgridp1;
  struct edge *  ep;

     length = 0;

     for (i = 0; i< nedges; i++) {
          ep = &grid[i];
          gridp1 = ep->p1;
          gridp2 = ep->p2;

          if ((gridp2 - gridp1)>1){
               gridp1 -= 1;
               Cgridp1 = gridp1;
               Cgridp1 %= N;
               gridp1 /= N;

              /*printf("C(%2d, %2d)\n", gridp1+1,Cgridp1+1);
               *Cout[gridp1][Cgridp1] += 1;
               */
               length += V[gridp1];
             }else
            {
               gridp1 -= 1;
               Rgridp1 = gridp1;
               Rgridp1 /= N;
               gridp1 %= N;

              /*printf("R(%2d, %2d)\n", Rgridp1+1,gridp1+1);
               *Rout[Rgridp1][gridp1] += 1;
               */
               length += H[gridp1];
             }
       }
       return(length);
   }
```

```
/**********************************
 *   delete degree-one-segments   *
 **********************************/

delete_degree_one(

struct edge *      edges,
int                pro_edge_count,
int *              degree_array,
int                save_redundant
)
{
int   i;
int   originalp;
int   candidatep1;
int   candidatep2;
int   remain;

       for (i = 0; i<N; i++){
            originalp = ((r[i]-1)*N + c[i]);
            degree_array[originalp] += 2;
       }

       for ( i=0; i<pro_edge_count; i++){
            candidatep1 = (edges + i)->p1;
            candidatep2 = (edges + i)->p2;
            degree_array[candidatep1] += 1;
            degree_array[candidatep2] += 1;
       }

       for (i=0; i<=N*N;i++)
             if (degree_array[i]== 0) degree_array[i] -= 1;

                /* printf("degree =\n");
               * for (i=1; i<=N*N; i++){
               *   printf("%3d\t", degree[i]);
               *      }
               * printf("\n");
               */


            do{
                  remain = 0;

                  for (i = 1; i<=N*N; i++){
                     if(degree_array[i] > 0){
                        if(degree_array[i] < 2){
                              pro_edge_count = delete_one(edges,
pro_edge_count, i, degree_array, save_redundant);
                              remain = 1;
                        }
                     }
                  }
```

```
            } while(remain > 0);

        if(save_redundant)
             redundant_edges -= num_redundant_edges;

        for (i=0; i<=N*N;i++)
              if (degree_array[i]== 0) degree_array[i] = 1;

              return (pro_edge_count);
}
```

```
/*****************************************
 *     delete one edge                   *
 *****************************************/


delete_one(

struct edge *     edges,
int               edge_count,
int               n,
int *             degree_list,
int               save_re
)
{
int     i;
int     j;
int     p1;
int     p2;

      for(i=0; i<edge_count; i++){
            if((edges+i) ->p1 != n && (edges+i)->p2 !=n)  continue;
            p1 = (edges+i) ->p1;
            p2 = (edges+i) ->p2;
            degree_list[p1] -= 1;
            degree_list[p2] -= 1;
            if(save_re) {
                  *(redundant_edges)= *(edges+i);
                  ++redundant_edges;
                  ++num_redundant_edges;
            }
            for(j=1; j<=edge_count - i; j++)
                  *(edges+i+j-1) = *(edges +i+j);

            --edge_count;
      }
        return(edge_count);
}
```

```
/*****************************************
 *     calculate PR and PC               *
 *****************************************/


PR_PC(

int i,
int j,
double * F,
double * G,
double * PR,
double * PC
)
{
int I;
int J;
int m;
int n;
int s;
int t;

    if (r[j]<r[i]) {
        I=r[j];
        n=r[i]-r[j];
        if (c[j]>c[i]){
            J=c[j];
            m=c[j]-c[i];
            for (s=I; s<=n+I; s++)
                for (t=J-m; t<=J-1; t++)
                     PR[(N+1)*s+t] += F[(N+1)*(m-J+t+1)+(n-
s+I)]*F[(N+1)*(J-t)+(s-I)]/F[(N+1)*(m+1)+n];
/*PR[s][t]+=F[m-J+t+1][n-s+I]*F[J-t][s-I]/F[m+1][n];*/

             for (s=I; s<=n+I-1; s++)
                for (t=J-m; t<=J; t++)
                     PC[(N+1)*s+t] += G[(N+1)*(m-J+t)+(n-
s+I)]*G[(N+1)*(J-t)+(s-I+1)]/G[(N+1)*m+(n+1)];
/*PC[s][t]+=G[m-J+t][n-s+I]*G[J-t][s-I+1]/G[m][n+1];*/
           }else
           {
               J=c[j];
               m=c[i]-c[j];
               for (s=I; s<=n+I; s++)
                   for (t=J; t<=m+J-1; t++)
                        PR[(N+1)*s+t] += F[(N+1)*(m-t+J)+(n-
s+I)]*F[(N+1)*(t-J+1)+(s-I)]/F[(N+1)*(m+1)+n];
/*PR[s][t]+=F[m-t+J][n-s+I]*F[t-J+1][s-I]/F[m+1][n];*/

               for (s=I; s<=n-1+I; s++)
                   for (t=J; t<=m+J; t++)
                        PC[(N+1)*s+t] += G[(N+1)*(m-t+J)+(n-
s+I)]*G[(N+1)*(t-J)+(s-I+1)]/G[(N+1)*m+(n+1)];
```

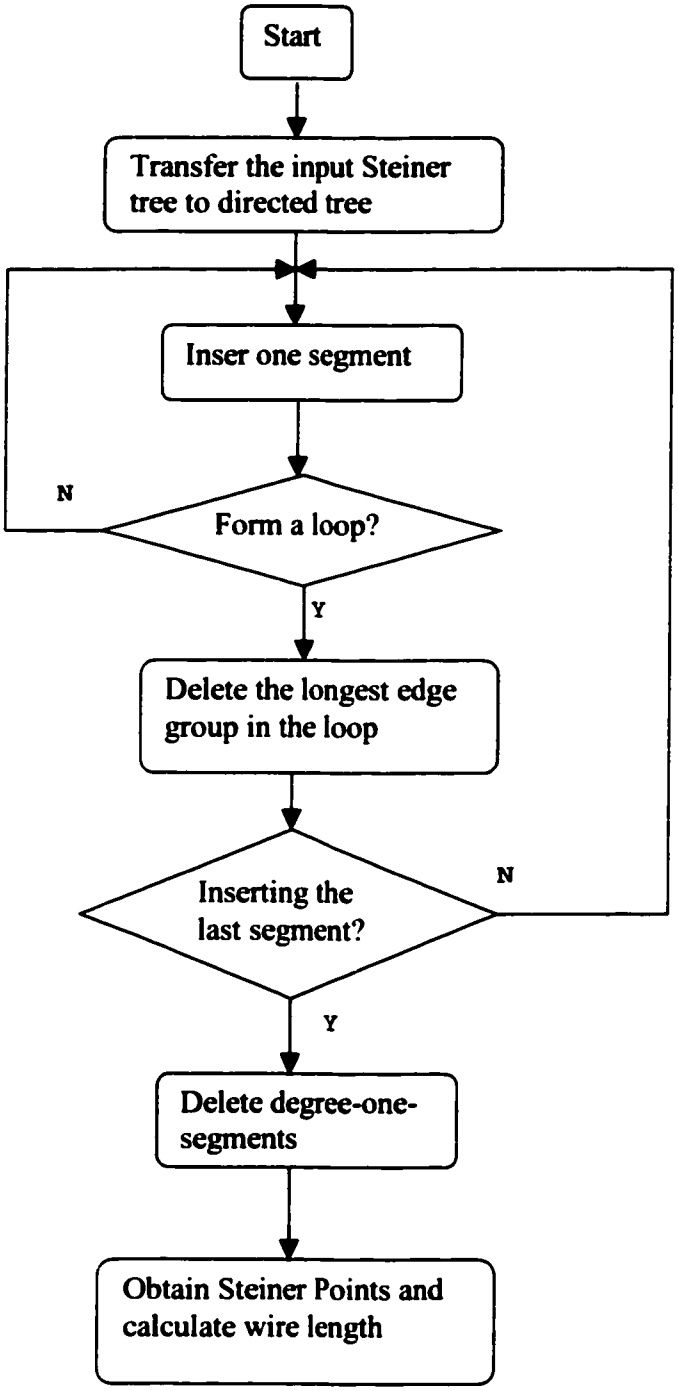
```
/*PC[s][t]+= G[m-t+J][n-s+I]*G[t-J][s-I+1]/G[m][n+1];*/
                }
       }else
      {

          I=r[i];
          n=r[j]-r[i];
          if (c[i]>c[j]){
              J=c[i];
              m=c[i]-c[j];
              for (s=I; s<=n+I; s++)
                  for (t=J-m; t<=J-1; t++)
                       PR[(N+1)*s+t] += F[(N+1)*(m-J+t+1)+(n-
s+I)]*F[(N+1)*(J-t)+(s-I)]/F[(N+1)*(m+1)+n];
/*PR[s][t]+=F[m-J+t+1][n-s+I]*F[J-t][s-I]/F[m+1][n];*/

              for (s=I; s<=n+I-1; s++)
                  for (t=J-m; t<=J; t++)
                      PC[(N+1)*s+t] += G[(N+1)*(m-J+t)+(n-
s+I)]*G[(N+1)*(J-t)+(s-I+1)]/G[(N+1)*m+(n+1)];
/*PC[s][t]+=G[m-J+t][n-s+I]*G[J-t][s-I+1]/G[m][n+1];*/
          }else
          {
               J=c[i];
               m=c[j]-c[i];
               for (s=I; s<=n+I; s++)
                   for (t=J; t<=m+J-1; t++)
                        PR[(N+1)*s+t] += F[(N+1)*(m-t+J)+(n-
s+I)]*F[(N+1)*(t-J+1)+(s-I)]/F[(N+1)*(m+1)+n];
/*PR[s][t]+= F[m-t+J][n-s+I]*F[t-J+1][s-I]/F[m+1][n];*/

               for (s=I; s<=n-1+I; s++)
                    for (t=J; t<=m+J; t++)
                        PC[(N+1)*s+t] += G[(N+1)*(m-t+J)+(n-
s+I)]*G[(N+1)*(t-J)+(s-I+1)]/G[(N+1)*m+(n+1)];
/*PC[s][t]+= G[m-t+J][n-s+I]*G[t-J][s-I+1] /G[m][n+1];*/
           }

      }
}
```

Appendix B

Source Code of Optimization Algotithm

This appendix contains the optimization algorithm source code. It starts with the results of probabilistic algorithms as well as any results of Steiner tree. It is edge-replacement operation, which introduces a new edge based on probabilities to the existing tree. If a loop is formed, then the longest edge or edge group is removed to break the loop. Therefore, the wire length is reduced. It displays the input Steiner tree result and the output Steiner tree result on the screen in graphic format, and shows the improvement of wire length.

The flow chart of optimization algorithm



```
/*************************************************************
*     transfer edge to node                                  *
*************************************************************/


make_node(

int * marked,
int   p1,
int   p2,
int   rowcol
)
{

 int  root;
 int  original;
 int  index1;
 int  index2;

      if (marked[p1]) {
            if (marked[p2]){
                  index1 = p1;
                  index2 = p2;
                  root = find_root(p1);
                  original = check_original(root);
                  if(!original) {
                        root = find_root(p2);
                        original = check_original(root);
                        if (original){
                              index1 = p2;
                              index2 = p1;
                        }
                  }
                  merge(index1, index2, rowcol);
            } else
            {
                  marked[p2] = 1;
                  root = find_root(p1);
                  original = check_original(root);
                  if (!original){
                        original = check_original(p2);
                        if (original){
                              Tree[p2].parent = NONE;
                              Tree[p2].weight = NONE;
                              merge(p2, p1, rowcol);
                              return;
                        }
                  }
                  Tree[p2].parent = p1;
                  /*printf("point %d, parent = %d\n", p2, p1);*/
```

```
                Tree[p2].weight = node_weight(p1, rowcol);
            }
        }else
        {
            if (marked[p2]) {
                marked[p1] = 1;
                root = find_root(p2);
                original = check_original(root);
                  if (!original){
                      original = check_original(p1);
                      if (original){
                            Tree[p1].parent = NONE;
                            Tree[p1].weight = NONE;
                            merge(p1, p2, rowcol);
                            return;
                      }
                }
                Tree[p1].parent = p2;
                Tree[p1].weight = node_weight(p1, rowcol);
            }else
            {
                marked[p1] = 1;
                marked[p2] = 1;
                index1 = p1;
                index2 = p2;
                original = check_original(p1);
                if(!original) {
                      original = check_original(p2);
                      if (original){
                            index1 = p2;
                            index2 = p1;
                      }
                }
                Tree[index1].parent = NONE;
                Tree[index1].weight = NONE;
                Tree[index2].parent = index1;
                Tree[index2].weight = node_weight(p1, rowcol);
            }
        }
}
```

```
/*************************************************************
*     merge two trees                                        *
*************************************************************/


merge (

int   index1,
int   index2,
int   rc
)
{
 int prev;
 int curr;
 int next;
 struct node preNode;
 struct node curNode;
 int tmp;

      prev = index2;
      curr = Tree[prev].parent;
      if (curr != NONE) {
            next = Tree[curr].parent;
            preNode = Tree[prev];
            curNode = Tree[curr];
            while ( 1 ){
                  Tree[curr]= preNode;
                  Tree[curr].parent = prev;
                  if (next == NONE) break;
                  preNode = curNode;
                  curNode = Tree[next];
                  prev= curr;
                  curr= next;
                  next= Tree[next].parent;
            }
      }
      Tree[index2].parent = index1;
      tmp = index1;
      if ( index1 > index2) tmp = index2;
      Tree[index2].weight = node_weight(tmp, rc);
}
```

```
/*************************************************************
*     calculate node weight                                  *
*************************************************************/


node_weight(

int   p1,
int   rc
)
{
  int        lp;
  double weight;

      if (rc) {
            lp = p1 - 1;
            lp /= N;
            weight = V[lp];
      }
      else {
            lp = p1 - 1;
            lp %= N;
            weight = H[lp];
      }
      return weight;
}
```

```
/*************************************************************
*     find the root of a tree from any point                 *
*************************************************************/


find_root(int i)
{
 int        j;
 int        k;

      j = Tree[i].parent;
      if (j == NONE) {
            return i;
      }
      while (TRUE) {
            k = Tree[j].parent;
            if (k== NONE) break;
            j = k;
      }
      return j;
}
```

```
/*************************************************************
*     check if the point is original                        *
*************************************************************/


check_original(int point)
{
 int i;
 int originalpoint;
 int yn;

      yn = 0;
      for (i = 0; i<N; i++){
            originalpoint = ((r[i]-1)*N + c[i]);
            if (point != originalpoint) continue;
            else yn = 1; return (yn);
       }
       return(yn);
}
```

```
/*********************************************************************
*     add the new edge to edge array and delete the longest one *
*********************************************************************/


update_edges(

int addp1,
int addp2,
int deletp,
int n
)
{
  int i, j, k, p1, p2;
  int curr;
  int next;

      degree[addp1] += 1;
      degree[addp2] += 1;
      (tree_edges+ num_tree_edges) ->p1 = addp1;
      (tree_edges+ num_tree_edges) ->p2 = addp2;
      ++num_tree_edges;
#ifdef DISPLAYOPTIMIZED
      printf("added edge(%d, %d)\n", addp1, addp2);
#endif
        curr = deletp;
      for (j = 1; j<=n; j++) {
            next = Tree[curr].parent;
            for (i = 0; i < num_tree_edges; i++){
                  p1 = (tree_edges+ i) ->p1;
                  p2 = (tree_edges+ i) ->p2;
                  if((p1 == curr && p2 == next)||(p2 == curr && p1 ==
next)) {
#ifdef DISPLAYOPTIMIZED
                        printf("deleted edge(%d, %d)\t", p1, p2);
#endif
                        degree[p1] -= 1;
                        degree[p2] -= 1;
                        for(k=1; k<=num_tree_edges - i; k++)
                              *(tree_edges+i+k-1) = *(tree_edges +i+k);
                        --num_tree_edges;
                  }
            }
            curr = next;
      }
      printf("\n");

}
```

```
/*************************************************************
*     calculate weight                                       *
*************************************************************/


 temp_move_up(int *index)
{
 int   temp;
 double weight;

      temp = *index;
      weight = Tree[temp].weight;
      if (Tree[temp].parent != NONE){
            weight = Tree[temp].weight;
            temp = Tree[temp].parent;

            if(degree[*index] == 1){
                  while(degree[temp]==1){
                        weight += Tree[temp].weight;
                        temp = Tree[temp].parent;
                        if(temp == NONE) break;
                  }
            }else
            {
                  while(degree[temp]==2){
                        weight += Tree[temp].weight;
                        temp = Tree[temp].parent;
                        if(temp == NONE) break;
                  }

            }
      }
      return (weight);
}
```

```
/*************************************************************
*     add_looped_edges                                       *
*************************************************************/


 add_looped_edges()
 {
  int              i;
  int              p1;
  int              p2;
  int              lp;
  double           length;
  struct edge*     ep;
  int              edge_count;
  int *            tmpdegree;

     delete_subtrees(); /*delete the trees which don't contain
                          original points*/

     id_num = 0;
     for (i = 0; i < num_loop_edges; i++) {  /* num_loop_edges*/
            ep = &loop_edges[i];
            p1 = ep -> p1;
            p2 = ep -> p2;
            if((p2 - p1) >1 ){
                  lp = p1 - 1;
                  lp /= N;
                  length = V[lp];
            }else
            {
                  lp = p2 - 2;
                  lp %= N;
                  length = H[lp];
            }
#ifdef DISPLAYOPTIMIZED
            printf("\ndegree[%d]=    %d,    degree[%d]=    %d\n",    p1,
degree[p1], p2, degree[p2]);
#endif
            if (degree[p1]> 0){
                  if (degree[p2]> 0)
                        add_edge(p1, p2, length);
                  else {
                        degree[p2] = 1;
                        Tree[p2].parent = p1;
                        Tree[p2].weight = length;
                  }
            }else if (degree[p2]> 0){
                        degree[p1] = 1;
                        Tree[p1].parent = p2;
                        Tree[p1].weight = length;
                  }
     }
```

```
    tmpdegree = NEWA(N*N + 1, int);
    for (i=0; i<=N*N;i++) tmpdegree[i]=0;
    edge_count = num_tree_edges;
    edge_count = delete_degree_one(tree_edges, num_tree_edges,
                                   tmpdegree, 0);

    free ((char*) tmpdegree);
    return edge_count;
}
```

```
/*************************************************************
*     real weight                                            *
*************************************************************/


 really_added_weight(

 int             point1,
 int             point2,
 double          length,
 int *           degreelist
 )
 {
  double         realweight;
  int            i;

     realweight = length;
     /*printf("p1 = %d, p2 = %d, len = %f\n", point1, point2,
weight);*/

     while(degree[point1] == 1){
            realweight += Tree[point1].weight;
            point1 = Tree[point1].parent;
            degreelist[point1] = 1;
            /*printf("p1 = %d, weight = %f\n", point1, realweight);*/

     }

      while(degree[point2] == 1){
            realweight += Tree[point2].weight;
            point2 = Tree[point2].parent;
            degreelist[point2] = 1;
            /*printf("p2 = %d, weight = %f\n", point2, realweight); */
     }

     for (i= 0; i<= N*N; i++)
     if (degreelist[i])
            degree[i] += 1;

     return realweight;
}
```

```
/*************************************************************
*     keep main tree and delete isalated subtrees           *
*************************************************************/


delete_subtrees()
{
 int *  main_sub;
 int    i, j;
 int    p1;
 int    p2;
 int    root;
 int    original;
 struct edge ep;

      main_sub = NEWA(N*N+1, int);
      for (i= 0; i<= N*N; i++) main_sub[i] = 0;

      for (i = 0; i < num_tree_edges; i++){
            p1 = (tree_edges+ i) ->p1;
            p2 = (tree_edges+ i) ->p2;

            root = find_root(p1);
            original = check_original(root);
            if(original) main_sub[p1] = 1;
            else         main_sub[p1] = -1;

            root = find_root(p2);
            original = check_original(root);
            if(original) main_sub[p2] = 1;
            else         main_sub[p2] = -1;
      }

      for (i= 0; i<= N*N; i++)
            if (main_sub[i]== -1) {
                  Tree[i].parent = 0;
                  Tree[i].weight = 0;
                  Tree[i].id     = 0;
                  degree[i]      = -1;
            }


      for(i=0; i<num_tree_edges; i++){
            p1 = (tree_edges+ i) ->p1;
            p2 = (tree_edges+ i) ->p2;

                  if(main_sub[p1]== -1 && main_sub[p2]== -1) {
                  ep = *(tree_edges+ i);
                  *(loop_edges+ num_loop_edges) = ep;
                  ++num_loop_edges;
                        for(j=1; j<= num_tree_edges - i; j++)
                        *(tree_edges+i+j-1) = *(tree_edges +i+j);
                  --num_tree_edges;
```

```
            }
        }

        free((char*) main_sub);
}
```

Vita Auctoris

Name:	Jiang Zhao	
Date of Birth:	15th May, 1968	
Place of Birth:	Shenyang, Liaoning, China	
Education:	1983-1986	Xinmin High School
	1986-1990	Wuhan University (B. S.)
	2000-2002	University of Windsor (M. A. Sc.)
Work History	1990-1997	Electrical Engineer, Shenyang Machine Tool Works
	1997-1999	Electrical Engineer, SYD Telecomunication Ltd.